Patent Office
Canberra

CERTIFIED COPY OF
PRIORITY DOCUMENT

I, JONNE YABSLEY, TEAM LEADER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. 2002953134 for a patent by SILVERBROOK RESEARCH PTY. LTD. as filed on 02 December 2002.

WITNESS my hand this
Ninth day of January 2004

JONNE YABSLEY
TEAM LEADER EXAMINATION
SUPPORT AND SALES

AUSTRALIA

Patents Act 1990

Provisional Specification

for an invention entitled:

METHOD AND APPARATUS (AUTH16)

The invention is described in the following statement: -

SoPEC

Security Overview

4-4-1-3 version 1.6

November 29, 2002

Silverbrook Research Pty Ltd
393 Darling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9818 6633
Fax: +61 2 9818 6711
Email: Info@SilverbrookResearch.com

1 Introduction

1.1 Document History

Version	Date	Authors	Details
1.6	29 November, 2002	Simon Walmsley	Updated ChipA to be ChipR to match protocols document, got rid of 68k reference now that we are using LEON.
1.5	26 November, 2002	Simon Walmsley	Added description of storing more than a single SoPEC_id key in a PRINTER_QA (in section 3.5.3 and related). This reduces the cost of a multi-SoPEC system with no loss of security. Also added text to describe that batch keys can be different for each SoPEC if the indirect upgrade key protocol is used.
1.4	9 September, 2002	Simon Walmsley	Added section in requirements detailing types of attacks we care about and don't care about.
1.3	30 August, 2002	Simon Walmsley	Changed ComCo_OEM_xxxx variables into simply xxxx variables, since that is more generic. Added text regarding ink refill. Added extra software authentication stage to prevent ComCos from fiddling with SoPEC software.
1.2	29 August, 2002	Simon Walmsley	Added section on how the PRINTER_QA chip gets programmed with the SoPEC_id_key.
1.1	28 August, 2002	Simon Walmsley	Updated to have ink and operating parameters be authenticated via symmetric key based signatures based on a unique SoPEC_id.
1.0	27 August, 2002	Simon Walmsley	Updated after review.
0.2 draft	26 August, 2002	Simon Walmsley	Changed public-key and private key references to asymmetric & symmetric respectively, so private can now sub-refer to the private key of the asymmetric pair, or the single private symmetric key. Changed OEM_Id into ComCo_OEM_license_id to more accurately reflect the scope of the id.
0.1 draft	26 August, 2002	Simon Walmsley	Initial issue.

1.2 References

[1] Silicon & Software Systems, *4-4-9-4 SoPEC Hardware Design.*
[2] Silverbrook Research, *4-2-1-1 Print Engine Controller Hardware Design.*
[3] Silverbrook Research, *4-3-1-2 QA Chip Technical Reference.*
[4] Silverbrook Research, *4-3-1-8 QA Chip Programmer Requirements.*
[5] Silverbrook Research, *4-3-1-26 Authentication Protocols.*

1.3 Scope

This document describes the basic security requirements of programs running on the SoPEC ASIC [1]. It then describes an implementation solution to the security requirements.

The described solution impacts the design of the SoPEC ASIC as well as implying key management issues. The solution includes references to the QA Chip ASIC [3] and associated authentication protocols [5].

It is possible that some of the requirements and defined solution will be applicable to systems built with the PEC ASIC [2], although such systems are beyond the scope of this document.

1.4 READERSHIP

This document is written for software engineers and system architects that are working with SoPEC, as well as PCB designers that are responsible for SoPEC-based Print Engines. A similar audience working on PEC and PEC-based Print Engines may also find document useful.

This document is also intended to be read by those responsible for key management and associated database designers with regards to guiding requirements.

This document is confidential to Silverbrook Research Pty. Ltd. and its distribution outside this organisation *must* be covered by a non-disclosure agreement (NDA).

1.5 QA CHIP TERMINOLOGY

The Authentication Protocols document [5] refers to QA Chips by their function in particular protocols:

- For authenticated reads, ChipR is the QA Chip being read from, and ChipT is the QA Chip that identifies whether the data read from ChipR can be trusted.
- For replacement of keys, ChipP is the QA Chip being programmed with the new key, and ChipF is the factory QA Chip that generates the message to program the new key.
- For upgrades of data in memory vectors, ChipU is the QA Chip being upgraded, and ChipS is the QA Chip that signs the upgrade value.

Any given physical QA Chip will contain functionality that allows it to operate as an entity in some number of these protocols.

Therefore, wherever the terms ChipR, ChipT, ChipP, ChipF, ChipU and ChipS are used in this document, they are referring to *logical* entities involved in an authentication protocol as defined in [5].

Physical QA Chips are referred to by their location. For example, each ink cartridge may contain a QA Chip referred to as an INK_QA, with all INK_QA chips being on the same physical bus. In the same way, the QA Chip inside the printer is referred to as PRINTER_QA, and will be on a separate bus to the INK_QA chips.

2 Requirements

2.1 SECURITY

The basic functional security requirements are:

- Silverbrook code and OEM program code co-existing safely
- Silverbrook operating parameters authentication
- OEM operating parameters authentication
- Ink usage authentication

Each of these is outlined in subsequent sections.

The authentication requirements imply that:

- OEMs and end-users must not be able to replace or tamper with Silverbrook program code or data
- OEMs and end-users must not be able to call unauthorized functions within Silverbrook code
- End-users must not be able to replace or tamper with OEM program code or data
- End-users must not be able to call unauthorized functions within OEM program code
- OEMs must be able to test products at their highest upgradable status, yet not be able to ship them outside the terms of their license
- OEMs and end-users must not be able to directly access the print engine pipeline (PEP) hardware, the LSS Master (for QA Chip access) or any other peripheral block with the exception of operating system permitted GPIO pins and timers.

2.1.1 Silverbrook code and OEM program code co-existing safely

SoPEC includes a CPU that must run both Silverbrook program code and OEM program code. The execution model envisaged for SoPEC is one where Silverbrook program code forms an operating system (O/S), providing services such as controlling the print engine pipeline, interfaces to communications channels etc. The OEM program code must run in a form of user mode, protected from harming the Silverbrook program code. The OEM program code is permitted to obtain services by calling functions in the O/S, and the O/S may also call OEM code at specific times. For example, the OEM program code may request that the O/S call an OEM interrupt service routine when a particular GPIO pin is activated.

A basic requirement then, for SoPEC, is a form of protection management, whereby Silverbrook and OEM program code can co-exist without the OEM program code damaging operations or services provided by the Silverbrook O/S. Since services rely on SoPEC peripherals (such as SCB, LSS Master, Timers etc) access to these peripherals should also be restricted to Silverbrook program code only.

2.1.2 Silverbrook operating parameters authentication

A particular OEM will be licensed to run a Print Engine with a particular set of operating parameters (such as print speed or quality). The OEM and/or end-user can upgrade the operating license for a fee and thereby obtain an upgraded set of operating parameters.

Neither the OEM nor end-user should be able to upgrade the operating parameters without paying the appropriate fee to upgrade the license. Similarly, neither the OEM nor end-user

should be able to bypass the authentication mechanism via any program code on SoPEC. This implies that OEMs and end-users must not be able to tamper with or replace Silverbrook program code or data, nor be able to call unauthorized functions within Silverbrook program code.

However, the OEM must be capable of assembly-line testing the Print Engine at the upgraded status before selling the Print Engine to the end-user.

2.1.3 OEM operating parameters authentication

The OEM may provide operating parameters to the end-user independent of the Silverbrook operating parameters. For example, the OEM may want to sell a franking machine[1].

The end-user should not be able to upgrade the operating parameters without paying the appropriate fee to the OEM. Similarly, the end-user should not be able to bypass the authentication mechanism via any program code on SoPEC. This implies that end-users must not be able to tamper with or replace OEM program code or data, as well as not be able to tamper with the PEP blocks or service-related peripherals.

2.1.4 Ink usage authentication

Each OEM sells printers and ink to end-users according to a business model. For example, OEM_1 may provide ink at $A for a $B printer, while OEM_2 may sell the same featured printer at a higher price $A+$X, and provide the ink at a cheaper price $B-$Y. OEM_1 has a business model that relies on the fact that end-users of OEM_1 printers can only use OEM_1 ink, and likewise OEM_2 has a business model that relies on the fact that end-users of OEM_2 printers can only use OEM_2 ink.

It is in the interest of both OEM_1 and OEM_2 that end-users cannot subvert the authentication mechanism for ink. Otherwise the business models are compromised.

It is also in the interests of the Memjet Group that OEM_1 and OEM_2 cannot subvert the authentication mechanism for ink, since the Memjet Group provides OEMs with printers under a license agreement that the OEM will purchase ink from a designated ink supplier.

2.2 Acceptable Compromises

Since there is no protection physically built into the Memjet printheads, it is theoretically possible for someone (with enough time, money and incentive) to remove the printheads from the print engine, build their own SoPEC ASIC equivalent, write their own program code etc. It is impossible to guard against such an attack.

We are really only concerned with commercial attacks, where there is a total compromise of printer operating parameter authentication and ink usage authentication. An example of such an attack is where the Silverbrook printing O/S is replaced by one that can be downloaded from the internet, and this clone O/S allows usage of the print engine outside the license agreement. Whether the clone O/S is developed by a hacker or by a rogue OEM is not important - it matters that any user can trivially upgrade the printer outside the terms of the license agreement.

1. a franking machine prints stamps

If an end user takes the time and energy to hack the print engine and thereby succeeds in upgrading the single print engine only, yet not be able to use the same keys etc on another print engine, that is an acceptable security compromise. However it doesn't mean we have to make it totally simple or cheap for the end-user to accomplish this.

Software-only attacks are the most dangerous, since they can be transmitted via the internet and have no perceived cost. Physical modification attacks are far less problematic, since most printer users are not likely to want their print engine to be physically modified. This is even more true if the cost of the physical modification is likely to exceed the price of a legitemate upgrade.

Finally, it should be noted that all OEMs are bound by license agreements that specify penalties if they attempt to reverse engineer or bypass the print engines. In countries where these agreements are enforceable by law, this at least provides a modicum of security.

2.3 IMPLEMENTATION CONSTRAINTS

Any solution to the requirements detailed in Section 2.1 must also meet certain implementation constraints. These are:

- No flash memory inside SoPEC
- SoPEC must be simple to verify
- Silverbrook program code must be updateable
- OEM program code must be updateable
- Must be bootable from activity on USB or ISI
- No extra pins for assigning IDs to slave SoPECs
- Cannot trust the comms channel to the QA Chip in the printer (PRINTER_QA)
- Cannot trust the comms channel to the QA Chip in the ink cartridges (INK_QA)
- Cannot trust the ISI comms channel

These constraints are detailed below.

2.3.1 No flash memory inside SoPEC

SoPEC is intended to be implemented in 0.13 micron or smaller. Flash memory will not be available in any of the target processes being considered. Although Virage have a process independent flash cell, it is very large and effectively impractical for anything more than a few bits.

2.3.2 SoPEC must be simple to verify

All combinatorial logic and embedded program code within SoPEC must be verified before manufacture. Every increase in complexity in either of these increases verification effort and increases risk.

2.3.3 Silverbrook program code must be updateable

It is not possible nor even desirable to write a single complete operating system that is:

- verified completely (see Section 2.3.1)
- correct for all possible future uses of SoPEC systems
- finished in time for SoPEC manufacture

Therefore the complete Silverbrook program code must not *permanently* reside on SoPEC. It must be possible to update the Silverbrook program code as enhancements to functionality are made and bug fixes are applied.

In the worst case, only new printers would receive the new functionality or bug fixes. In the best case, existing SoPEC users can download new embedded code to enable functionality or bug fixes. Ideally, these same users would be obtaining these updates from the OEM website or equivalent, and not require any interaction with Silverbrook.

2.3.4 OEM program code must be updateable

Given that each OEM will be writing specific program code for printers that have not yet been conceived, it is impossible for all OEM program code to be embedded in SoPEC at the ASIC manufacture stage.

Since flash memory is not available (see Section 2.3.1), OEMs cannot store their program code in on-chip flash. While it is theoretically possible to store OEM program code in ROM on SoPEC, this would entail OEM-specific ASICs which would be prohibitively expensive. Therefore OEM program code cannot *permanently* reside on SoPEC.

Since OEM program code must be downloadable for SoPEC to execute, it should therefore be possible to update the OEM program code as enhancements to functionality are made and bug fixes are applied.

In the worst case, only new printers would receive the new functionality or bug fixes. In the best case, existing SoPEC users can download new embedded code to enable functionality or bug fixes. Ideally, these same users would be obtaining these updates from the OEM website or equivalent, and not require any interaction with Silverbrook.

2.3.5 Must be bootable from activity on USB or ISI

SoPEC can be placed in sleep mode to save power when printing is not required. RAM is not preserved in sleep mode. Therefore any program code and data in RAM will be lost. However, SoPEC must be capable of being woken up from the host when it is time to print again.

In the case of a single SoPEC system, the host communicates with SoPEC via USB.

In the case of a multi-SoPEC system, the host typically communicates with the ISI Master chip (e.g. the ISI Master could be SoPEC, and the comms is USB), and can send messages to other slave SoPECs via the ISI master. The ISI master SoPEC relays these messages to the slaves via the ISI.

Therefore SoPEC must be capable of being woken up by activity on either the USB or on the ISI.

2.3.6 No extra pins to assign IDs to slave SoPECs

In a single SoPEC system the host only sends data to the single SoPEC. However in a multi-SoPEC system, each of the slaves needs to be uniquely identifiable in order to be able for the host to send data to the correct slave.

Since there is no flash on board SoPEC (Section 2.3.1) we are unable to store a slave ID (eg 4 bits) in each SoPEC. Moreover, any ROM in each SoPEC will be identical.

It is possible to assign n pins to allow 2^n combinations of IDs for slave SoPECs. However a design goal of SoPEC is to minimize pins for cost reasons, and this is particularly true of features only used in multi-SoPEC systems. We have 2 pins for inter-SoPEC communications, and further pins would add to the cost.

The design constraint requirement is therefore to allow slaves to be IDed via a method that does not require any extra pins. This implies that whatever boot mechanism that satisfies the security requirements of Section 2.1 must also be able to assign IDs to slave SoPECs.

2.3.7 Cannot trust the comms channel to the QA Chip in the printer (PRINTER_QA)

If the printer operating parameters are stored in the non-volatile memory of the Print Engine's on-board PRINTER_QA chip, both Silverbrook and OEM program code cannot rely on the communication channel being secure. It is possible for an end-user to replace the PRINTER_QA chip or subvert the communications channel.

2.3.8 Cannot trust the comms channel to the QA Chip in the ink cartridges (INK_QA)

The amount of ink remaining for a given ink cartridge is stored in the non-volatile memory of that ink cartridge's INK_QA chip. Both Silverbrook and OEM program code cannot rely on the communication channel to the INK_QA being secure. It is possible for an end-user to replace the INK_QA chip or subvert the communications channel.

2.3.9 Cannot trust the ISI comms channel

In a multi-SoPEC system, or in a single-SoPEC system that has a non-USB connection to the host, a given SoPEC will receive its data over the ISI. It is quite possible for an end-user to insert a chip that subverts the communications channel (for example performs man-in-the-middle attacks).

3 Proposed Solution

A proposed solution to the requirements of Section 2, can be summarised as:

- Each SoPEC has a unique id
- CPU with user/supervisor mode
- Memory Management Unit
- Specific entry points in O/S
- Boot procedure, including authentication of program code and operating parameters
- SoPEC ISI identification

3.1 EACH SOPEC HAS A UNIQUE ID

Each SoPEC needs to contains a unique *SoPEC_id* of minimum size 64-bits[1]. This *SoPEC_id* is used to form a symmetric key unique to each SoPEC: *SoPEC_id_key*.

The verification of operating parameters and ink usage depends on *SoPEC_id* being difficult to determine. Difficult to determine means that someone should not be able to determine the id via software, or by viewing the communications between chips on the board. If the *SoPEC_id* is available through running a test procedure on specific test pins on the chip, then depending on the ease by which this can be done, it is likely to be acceptable.

It is important to note that in the proposed solution, compromise of the *SoPEC_id* leads only to compromise of the operating parameters and ink usage on this particular SoPEC. It does not compromise any other SoPEC or all inks or operating parameters in general.

It is ideal that the *SoPEC_id* be random, although this is unlikely to occur on standard manufacture processes for ASICs. If the id is within a small range however, it will be able to be broken by brute force. This is why 32-bits is not sufficient protection.

3.2 CPU WITH USER/SUPERVISOR MODE

SoPEC contains a CPU with direct hardware support for user and supervisor modes. At present, the intended CPU is the LEON (a 32-bit processor with an instruction set according to the IEEE-1754 standard. The IEEE1754 standard is compatible with the SPARC V8 instruction set).

Silverbrook (operating system) program code will run in supervisor mode, and all OEM program code will run in user mode.

3.3 MEMORY MANAGEMENT UNIT

SoPEC contains a Memory Management Unit (MMU) that limits access to regions of DRAM by defining read, write and execute access permissions for supervisor and user mode. Program code running in user mode is subject to user mode permission settings, and program code running in supervisor mode is subject to supervisor mode settings.

1. On IBM's CU11 process this chipId is 80 bits.

A setting of 1 for a permission bit means that type of access (e.g. read, write, execute) *is* permitted. A setting of 0 for a read permission bit means that that type of access is *not* permitted.

At reset and whenever SoPEC wakes up, the settings for all the permission bits are 1 for all supervisor mode accesses, and 0 for all user mode accesses. This means that supervisor mode program code must explicitly set user mode access to be permitted on a section of DRAM.

Access permission to all the non-valid address space should be trapped, regardless of user or supervisor mode, and regardless of the access being read, execute, or write.

Access permission to all of the valid non-DRAM address space (for example the PEP blocks) is supervisor read / write access only (no supervisor execute access, and user mode has no acccess at all) with the exception that certain GPIO and Timer registers can also be accessed by user code. These registers will require bitwise access permissions. Each peripheral block will determine how the access is restricted.

The embedded DRAM should start at 0x0000_0000 to support programmable exception vectors. The reset exception vector (and possibly some others) is translated in the MMU to point to the appropriate location in ROM, ideally in a manner that still allows null pointer dereferencing to be trapped.

With respect to the DRAM and PEP subsystems of SoPEC, typically we would set user read/write/execute mode permissions to be 1/1/0 only in the region of memory that is used for OEM program data, 1/0/1 for regions of OEM program code, and 0/0/0 elsewhere. By contrast we would typically set supervisor mode read/write/execute permissions for this memory to be 1/1/0 (to avoid accidentally executing user code in supervisor mode).

The *SoPEC_id* parameter (see Section 3.1) should only be accessible in supervisor mode, and should only be stored and manipulated in a region of memory that has no user mode access.

3.4 SPECIFIC ENTRY POINTS IN O/S

Given that user mode program code cannot even call functions in supervisor code space, the question arises as how OEM programs can access functions, or request services. The implementation for this depends on the CPU.

On the LEON processor, the TRAP instruction allows programs to switch between user and supervisor mode in a controlled way. The TRAP switches between user and supervisor register sets, and calls a specific entry point in the supervisor code space in supervisor mode. The TRAP handler dispatches the service request, and then returns to the caller in user mode.

Use of a command dispatcher allows the O/S to provide services that filter access - e.g. a generalised print function will set PEP registers appropriately and ensure QA Chip ink updates occur.

The LEON also allows supervisor mode code to call user mode code. There are a number of ways that this functionality can be implemented.

3.5 BOOT PROCEDURE

3.5.1 Basic premise

The intention is to load the Silverbrook and OEM program code down into SoPEC's RAM, where it can be subsequently executed. The basic SoPEC therefore, must be capable of downloading program code. However SoPEC must be able to guarantee that only authorized Silverbrook boot programs can be downloaded, otherwise anyone could modify the O/S to do anything, and then download that - thereby bypassing the licensed operating parameters.

We perform authentication of program code and data using asymmetric cryptography and *without* using a QA Chip.

Assuming we have already downloaded some data and a 160-bit signature into eDRAM, the boot loader needs to perform the following tasks:

- perform SHA-1 on the downloaded data to calculate a digest *localDigest*
- perform asymmetric decryption on the downloaded signature (160-bits) using an asymmetric public key to obtain *authorizedDigest*
- If *localDigest* = *authorizedDigest*, then the downloaded data is authorized (the signature must have been signed with the asymmetric private key) and control can then be passed to the downloaded data

Asymmetric decryption is used instead of symmetric decryption because the decrypting key must be held in SoPEC's ROM. If symmetric private keys are used, the ROM can be probed and the security is compromised.

The procedure requires the following data item:

- **boot0key** = an *n*-bit asymmetric public key

The procedure also requires the following two functions:

- **SHA-1**= a function that performs SHA-1 on a range of memory and returns a 160-bit digest
- **decrypt** = a function that performs asymmetric decryption of a message using the passed-in key

Assuming that all of these are available (e.g. in the boot ROM), boot loader 0 can be defined as in the following pseudocode:

```
bootloader0(data, sig)
   localDigest ← SHA-1(data)
   authorizedDigest ← decrypt(sig, boot0key)
   if (localDigest = authorizedDigest)
      jump to program code at data-start address// will never to return
   Else
      // program code is unauthorized
   EndIf
```

The length of the key will depend on the asymmetric algorithm chosen. The key must provide the equivalent protection of the entire QA Chip system - if the Silverbrook O/S program code can be bypassed, then it is equivalent to the QA Chip keys being compromised. In fact it is worse because it would compromise Silverbrook operating parameters, OEM operating parameters, and ink authentication by software downloaded off the net (e.g. from some hacker in Norway).

In the case of RSA, a 2048-bit key is required to match the 160-bit symmetric-key security of the QA Chip. In the case of ECDSA, a key length of 132 bits is likely to suffice.

There is also no advantage to storing multiple keys in SoPEC and having the *external* message choose which key to validate against, because a compromise of any key allows the external user to always select that key. Even if there were 2 keys, one for USB-based booting and one for ISI-based booting, a compromise of the USB-based booting key is enough to compromise all the single SoPEC systems (99% of SoPEC systems).

Therefore the entire security of SoPEC is based on keeping the asymmetric private key paired to boot0key secure. The entire security of SoPEC is also based on keeping the program that signs (i.e. authorizes) datasets using the asymmetric private key paired to boot0key secure.

If a compromise is discovered, it may be economically feasible to change the *boot0key* value in SoPEC's ROM, since this is only a single mask change, and would be easy to verify and characterize.

3.5.2 Hierarchies of authentication

Given that test programs, evaluation programs, and Silverbrook O/S code needs to be written and tested, and OEM program code etc. also needs to be tested, it is not secure to have a single authentication of a monolithic dataset combining Silverbrook O/S, non-O/S, and OEM program code - we certainly don't want OEM's signing Silverbrook program code, and Silverbrook shouldn't have to be involved with the signing of OEM program code.

Therefore we require differing levels of authentication and therefore a number of keys, although the procedure for authentication is identical to the first - a section of program code contains the key for authenticating the next.

This method allows for any hierarchy of authentication, based on a root key of boot0key.

For example, assume that we have the following entities:

- SoPECCo, Silverbrook's SoPEC hardware / software company. Supplies SoPEC ASICs and SoPEC O/S printing software to a ComCo.
- ComCo, a company that assembles Print Engines from SoPECs, Memjet printheads etc, customizing the Print Engine for a given OEM according to a license
- OEM, a company that uses a Print Engine to create a printer product to sell to the end-users. The OEM would supply the motor control logic, user interface, and casing.

The levels of authentication hierarchy are as follows:

- SoPECCo generates *dataset1*, consisting of the print engine O/S (which incorporates the print engine functionality) and the ComCo's asymmetric public key. SoPECCo signs *dataset0* with SoPECCo's asymmetric private *boot0key* key. The print engine program code expects to see an operating parameter block signed by the ComCo's asymmetric private key.
- The ComCo generates *dataSet3*, consisting of *dataset1* plus *dataset2*, where *dataset2* is an operating parameter block for a given OEM's print engine licence (according to the print engine license arrangement) signed with the ComCo's asymmetric private key. The operating parameter block (*dataset2*) would contain valid print speed ranges, a *PrintEngineLicenseId*, and the OEM's asymmetric public key. The ComCo can gen-

erate as many of these operating parameter blocks for any number of Print Engine Licenses, but cannot write or sign any supervisor O/S program code.

- The OEM would generate *dataset5*, consisting of *dataset3* plus *dataset4*, where *dataset4* is the OEM program code signed with the OEM's asymmetric private key. The OEM can produce as many versions of *dataset5* as it likes (e.g. for testing purposes or for updates to drivers etc)

The relationship is shown below in Figure 1.

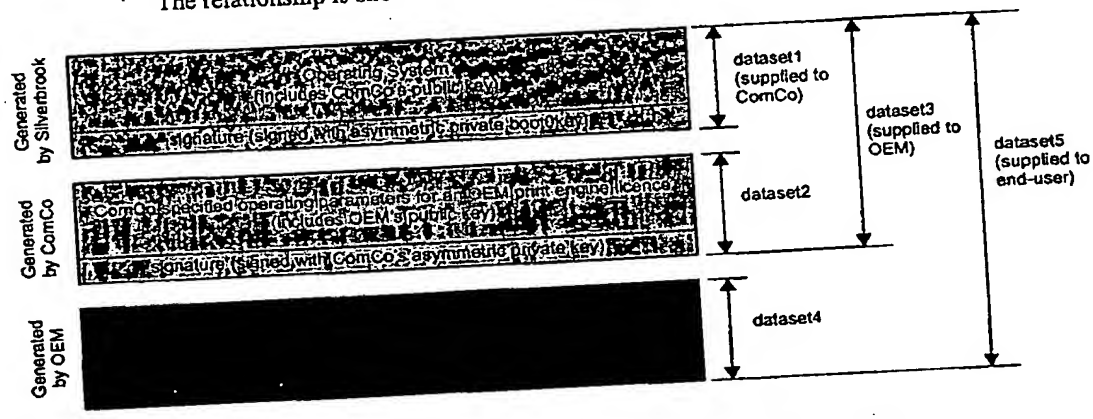


Figure 1. Relationship between the datasets

When the end-user uses *dataset5*, SoPEC itself validates *dataset1* via the *boot0key* mechanism described in Section 3.5.1. Once *dataset1* is executing, it validates *dataset2*, and uses *dataset2* data to validate *dataset4*. The validation hierarchy is shown in Figure 2.

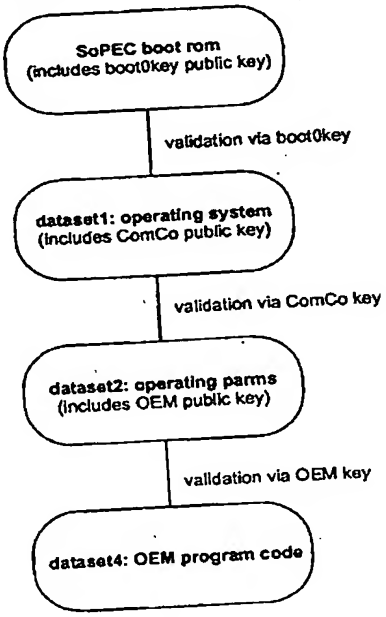


Figure 2. Validation hierarchy

If a key is compromised, it compromises all subsequent authorizations down the hierarchy. In the example from above (and as illustrated in Figure 2) if the OEM's asymmetric private key is compromised, then O/S program code is not compromised since it is above OEM program code in the authentication hierarchy. However if the ComCo's asymmetric private key is compromised, then the OEM program code is also compromised. A compromise of *boot0key* compromises everything up to SoPEC itself, and would require a mask ROM change in SoPEC to fix.

It is worthwhile repeating that in any hierarchy the security of the entire hierarchy is based on keeping the asymmetric private key paired to boot0key secure. It is also a requirement that the program that signs (i.e. authorizes) datasets using the asymmetric private key paired to boot0key secure.

3.5.3 Authenticating operating parameters

Operating parameters need to be considered in terms of Silverbrook operating parameters and OEM operating parameters. Both sets of operating parameters are stored on the PRINTER_QA chip (physically located inside the printer). This allows the printer to maintain parameters regardless of being moved to different computers, or a loss/replacement of host O/S drivers etc.

On PRINTER_QA, memory vector M_0 contains the upgradable operating parameters, and memory vectors M_{1+} contains any constant (non-upgradable) operating parameters.

Considering only Silverbrook operating parameters for the moment, there are actually two problems:

a. setting and storing the Silverbrook operating parameters, which should be authorized only by Silverbrook

b. reading the parameters into SoPEC, which is an issue of SoPEC authenticating the data on the PRINTER_QA chip since we don't trust PRINTER_QA.

The PRINTER_QA chip therefore contains the following symmetric keys:

- K_0 = *PrintEngineLicense_key*. This key is constant for all SoPECs supplied for a given print engine license agreement between an OEM and a Silverbrook ComCo. K_0 has write permissions to the Silverbrook upgradeable region of M_0 on PRINTER_QA.
- K_1 = *SoPEC_id_key*. This key is unique for each SoPEC (see Section 3.1), and is known only to the SoPEC and PRINTER_QA. K_1 does not have write permissions for anything.

K_0 is used to solve problem (a). It is only used to authenticate the actual upgrades of the operating parameters. Upgrades are performed using the standard upgrade protocol described in [5], with PRINTER_QA acting as the ChipU, and the external upgrader acting as the ChipS.

K_1 is used by SoPEC to solve problem (b). It is used to authenticate reads of data (i.e. the operating parameters) from PRINTER_QA. The procedure follows the standard authenticated read protocol described in [5], with PRINTER_QA acting as ChipR, and the embedded supervisor software on SoPEC acting as ChipT. The authenticated read protocol [5] requires the use of a 160-bit nonce, which is a pseudo-random number. This creates the problem of introducing pseudo-randomness into SoPEC that is not readily determinable by OEM programs, especially given that SoPEC boots into a known state. One possibility is to use the same random number generator as in the QA Chip (a 160-bit maximal-lengthed linear feedback shift register) with the seed taken from the value in the *WatchDogTimer* register in SoPEC's timer unit when the first page arrives.

Note that the procedure for verifying reads of data from PRINTER_QA does not rely on Silverbrook's key K_0. This means that precisely the same mechanism can be used to read and authenticate the OEM data also stored in PRINTER_QA. Of course this must be done by Silverbrook supervisor code so that *SoPEC_id_key* is not revealed.

If the OEM also requires upgradable parameters, we can add an extra key to PRINTER_QA, where that key is an OEM_key and has write permissions to the OEM part of M_0.

In this way, K_1 never needs to be known by anyone except the SoPEC and PRINTER_QA.

Each printing SoPEC in a multi-SoPEC system need access to a PRINTER_QA chip that contains the appropriate *SoPEC_id_key* to validate ink useage and operating parameters. This can be accomplished by a separate PRINTER_QA for each SoPEC, or by adding extra keys (multiple *SoPEC_id_keys*) to a single PRINTER_QA.

However, if ink usage is not being validated (e.g. if print speed were the only Silverbrook upgradable parameter) then not all SoPECs require access to a PRINTER_QA chip that contains the appropriate *SoPEC_id_key*. Assuming that OEM program code controls the physical motor speed (different motors per OEM), then the PHI within the first (or only) front-page SoPEC can be programmed to accept (or generate) line sync pulses at a particular rate. If line syncs arrived faster than the particular rate, the PHI would generate a buffer underrun. This would mean that even if the motor speed was hacked to be fast, the print will terminate.

3.5.3.1 OEM assembly-line test

As described in Section 2.1.2, Silverbrook operating parameters include such items as print speed, print quality etc. and are tied to a license provided to an OEM. These parameters are under Silverbrook control. The licensed Silverbrook operating parameters are stored in the PRINTER_QA as described in Section 3.5.3.

However, although an OEM should only be able sell the licensed operating parameters for a given Print Engine, they must be able to assembly-line test[1] the Print Engine with a different set of operating parameters i.e. a maximally upgraded Print Engine.

Several different mechanisms can be employed to allow OEMs to test the upgraded capabilities of the Print Engine. At present it is unclear exactly what kind of assembly-line tests would be performed.

At first thought, it might be considered that a dongle-style approach using a special master PRINTER_QA containing upgraded parameters might be a solution. However, for the SoPEC to accept the parameters as true, the special master PRINTER_QA must contain the appropriate *SoPEC_id_key* (tied to the specific *SoPEC_id* of the SoPEC system under test). Since the OEM can't perform the key upgrade, they must make use of a Silverbrook upgrade mechanism, which implies either a Silverbrook box, or a connection to a Silverbrook machine (e.g. over a net). Neither approaches are good.

1. This section is referring to assembly-line testing rather than development testing. An OEM can maximally upgrade a given Print Engine to allow developmental testing of their own OEM program code & mechanics.

If there is no special master PRINTER_QA for testing, then we must make use of special test programs, or storage on the PRINTER_QA, or both. The solution will depend on the test requirements of the OEM.

A simple test program that allows any pages to be printed at full upgrade capability is definitely not secure. If it gets out to the public, it is effectively a free upgrade. Silverbrook would not want the OEM to have such a program.

Likewise, if a test program only printed pages that had been signed with some key, then not only does this change the timing of real pages (since the signature must be verified before printing) but a service must be made available to sign special test images. This may be possible, but it means that Silverbrook must be involved for every time a test image is designed. If Silverbrook gives the OEM a program that generates signatures to avoid this annoyance, then it is the same as giving away the ability to print at full capability.

If the OEM requires tests that are not actually printing dots, then a test harness software loader that looks and behaves like a real Print Engine (including all output signals etc.) at full upgrade capability, except that it does not produce physical dots (i.e. at the very end of the pipeline, the data is masked before being sent out to the printhead). This will produce a timing accurate result, and is the simplest, most effective solution. If the special driver gets out into the public, the user can only print blank pages.

If the OEM requires tests that actually prints dots, there are several possibilities:

a. A version of the O/S (signed for the OEM) that will only print special Silverbrook test patterns. This may be quite adequate, but it has the disadvantage that OEM test patterns cannot be printed.

b. A version of the O/S that prints garbage in special places over the test image. Again the has the disadvantage that special OEM test patterns cannot be printed.

c. A version of the O/S that reads and decrements a DecrementOnly value in PRINTER_QA. If the value before successful decrementing is non-zero, then the O/S will run at full upgrade capability until either a power-loss or a pre-determined number of pages (e.g. agreed to by the OEM and Silverbrook) have been printed. The number to be stored in the PRINTER_QA at initial PRINTER_QA customization may only need to be 1 or 2.

Of these solutions, option (c) is probably the least restrictive to the OEM while still being useful. If the test program gets out, then if the value in PRINTER_QA is 0 after testing, then there is no impact, and if the value is small, then only a small number of pages can be printed at full upgrade capability, and power must stay on while doing so.

3.5.4 Use of a PrintEngineLicense Id

Silverbrook O/S program code contains the OEM's asymmetric public key to ensure that the subsequent OEM program code is authentic - i.e. from the OEM. However given that SoPEC only contains a single root key, it is theoretically possible for different OEM's applications to be run identically *physical* Print Engines i.e. printer driver for OEM_1 run on an identically *physical* Print Engine from OEM_2.

To guard against this, the Silverbrook O/S program code contains a *PrintEngineLicense_id* code (e.g. 16 bits) that matches the same named value stored as a fixed operating parameter in the PRINTER_QA (i.e. in M_{1+}). As with all other operating parameters, the value of *PrintEngineLicense_id* would be stored in PRINTER_QA at the

same time as the other various PRINTER_QA customizations are being applied, before being shipped to the OEM site.

In this way, the OEMs can be sure of differentiating themselves through software functionality.

3.5.5 Authentication of ink

The Silverbrook O/S must perform ink authentication [5] during prints. Ink usage authentication makes use of counters in SoPEC that keep an accurate record of the exact number of dots printed for each ink.

The ink amount remaining in a given cartridge is stored in that cartridge's INK_QA chip. Other data stored on the INK_QA chip includes ink color, viscosity, Memjet firing pulse profile information, as well as licensing parameters such as OEM_Id, inkType, InkUsageLicense_Id, etc. This information is typically constant, and is therefore likely to be stored in M_{1+} within INK_QA.

Just as the Print Engine operating parameters are validated by means of PRINTER_QA, a given Print Engine license may only be permitted to function with specifically licensed ink. Therefore the software on SoPEC could contain a valid set of ink types, colors, OEM_Ids, InkUsageLicense_Ids etc. for subsequent matching against the data in the INK_QA.

SoPEC must be able to authenticate reads from the INK_QA, both in terms of ink parameters as well as ink remaining.

To authenticate ink a number of steps must be taken:

- restrict access to dot counts
- authenticate ink usage and ink parameters via INK_QA and PRINTER_QA
- broadcast ink dot usage to all SoPECs in a multi-SoPEC system

3.5.5.1 restrict access to dot counts

Since the dot counts are accessed via the PHI in the PEP section of SoPEC, access to these registers (and more generally *all* PEP registers) must be only available from supervisor mode, and not by OEM code (running in user mode). Otherwise it might be possible for OEM program code to clear dot counts before authentication has occurred.

3.5.5.2 authenticate ink usage and ink parameters via INK_QA and PRINTER_QA

The basic problem of authentication of ink remaining and other ink data boils down to the problem that we don't trust INK_QA. Therefore how can a SoPEC know the initial value of ink (or the ink parameters), and how can a SoPEC know that after a write to the INK_QA, the count has been correctly decremented.

Taking the first issue, which is determining the initial ink count or the ink parameters, we need a system whereby a given SoPEC can perform an authenticated read of the data in INK_QA.

We cannot write the *SoPEC_id_key* to the INK_QA for two reasons:

- updating keys is not power-safe (i.e. if power is removed mid-update, the INK_QA could be rendered useless)

- the ink cartridge would then not work in another printer since the other printer would not know the old *SoPEC_id_key* (knowledge of the old key is required in order to change the old key to a new one).

The proposed solution is to let INK_QA have two keys:

- K_0 = *SupplyInkLicense_key*. This key is constant for all ink cartridges for a given ink supply agreement between an OEM and a Silverbrook ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_0 has write permissions to the ink remaining regions of M_0 on INK_QA.
- K_1 = *UseInkLicense_key*. This is key is constant for all ink cartridges for a given ink usage agreement between an OEM and a Silverbrook ComCo (this is *not* the same key as *PrintEngineLicense_key* which is stored as K_0 in PRINTER_QA). K_1 has no write permissions to anything.

K_0 is used to authenticate the actual upgrades of the amount of ink remaining (e.g. to fill and refill the amount of ink). Upgrades are performed using the standard upgrade protocol described in [5], with INK_QA acting as the ChipU, and the external upgrader acting as the ChipS. The fill and refill upgrader (ChipS) also needs to check the appropriate ink licensing parameters such as OEM_Id, InkType and InkUsageLicense_Id for validity.

K_1 is used to allow SoPEC to authenticate reads of the ink remaining and any other ink data. This is accomplished by having the same *UseInkLicense_key* within PRINTER_QA (e.g. in K_2), also with no write permissions.

This means there are two shared keys, with PRINTER_QA sharing both, and thereby acting as a bridge between INK_QA and SoPEC.

- *UseInkLicense_key* is shared between INK_QA and PRINTER_QA
- *SoPEC_id_key* is shared between SoPEC and PRINTER_QA

All SoPEC has to do is do an authenticated read [5] from INK_QA, pass the data / signature to PRINTER_QA, let PRINTER_QA validate the data / signature, and then get PRINTER_QA to produce a similar signature based on the shared *SoPEC_id_key*. SoPEC can then compare PRINTER_QA's signature with its own calculated signature (i.e. implement a Test function [5] in software on the SoPEC), and if the signatures match, the data from INK_QA must be valid, and can therefore be trusted.

Once the data from INK_QA is known to be trusted, the amount of ink remaining can be checked, and the other ink licensing parameters such as OEM_Id, InkType, InkUsageLicense_Id can be checked for validity.

The actual steps of read authentication as performed by SoPEC are:

```
KEY1 ← 1 // simple constants to specify which key to use when signing
KEY2 ← 2
R_PRINTER ← PRINTER_QA.random()
R_INK, M_INK, SIG_INK ← INK_QA.read(KEY1, R_PRINTER)// read with key1: UseInkLicense_key
R_SOPEC ← random()
SIG_PRINTER ← PRINTER_QA.test(KEY2, R_INK, M_INK, SIG_INK, KEY1, R_SOPEC)
SIG_SOPEC ← HMAC_SHA_1(R_PRINTER | R_SOPEC | M_INK)
If ((SIG_PRINTER != 0) AND (SIG_PRINTER = SIG_SOPEC))
   // M_INK (data read from INK_QA) is valid
   // M_INK could be ink parameters, such as InkUsageLicense_Id, or ink remaining
   If (M_INK.inkRemaining = expectedInkRemaining)
      // all is ok
   Else
```

```
      // the ink value is not what we wrote, so don't print anything anymore
   EndIf
Else
   // the data read from INK_QA is not valid and cannot be trusted
EndIf
```

Strictly speaking, we don't need a nonce (R_{SOPEC}) all the time because M_A (containing the ink remaining) should be decrementing between authentications. However we do need one to retrieve the initial amount of ink and the other ink parameters (at power up). This is why taking a random number from the *WatchDogTimer* at the receipt of the first page is acceptable.

In summary, the SoPEC performs the non-authenticated write [5] of ink remaining to the INK_QA chip, and then performs an authenticated read of the data via the PRINTER_QA as per the pseudocode above. If the value is authenticated, *and* the INK_QA ink-remaining value matches the expected value, the count was correctly decremented and the printing can continue.

3.5.5.3 broadcast ink dot usage to all SoPECs in a multi-SoPEC system

In a multi-SoPEC system, each SoPEC *attached to a printhead* (4 at most) must broadcast its ink usage to all the SoPECs. In this way, each SoPEC will have its own version of the expected ink usage.

In the case of a man-in-the-middle attack, at worst the count in a given SoPEC is only its own count (i.e. all broadcasts are turned into 0 ink usage by the man-in-the-middle).

A single SoPEC performs the update of ink remaining to the INK_QA chip, and then all SoPECs perform an authenticated read of the data via the appropriate PRINTER_QA (the PRINTER_QA that contains their matching *SoPEC_id_key* - remember that multiple *SoPEC_id_keys* can be stored in a single PRINTER_QA). If the value is authenticated, and the INK_QA value matches the expected value, the count was correctly decremented and the printing can continue.

If any of the broadcasts are not received, or have been tampered with, the updated ink counts will not match. The only case this does not cater for is if each SoPEC is tricked (via an ISI man-in-the-middle attack). into a total that is the same, yet not the true total. Apart from the fact that this is not viable for general pages, at worst this is the maximum amount of ink printed by a single SoPEC. We don't care about protecting against this case.

Since there will be at most 4 printing SoPEC, it requires at most 4 authenticated reads. This should be completed within 0.5 seconds - well within the 2 seconds/page print time.

3.5.6 Example hierarchy

The exact breakdown of hierarchy will depend on a later investigation, but for the purposes of scoping out possibilities, it is worthwhile considering an example hierarchy for illustrative purposes.

Adding an extra bootloader step to the example from Section 3.5.2, we can break up the contents of program space into logical sections, as shown in Table 1. Note that the ComCo does not provide any program code, merely operating parameters that is used by the O/S.

Table 1. Sections of Program Space

section	contents	verifies
0 (ROM)	boot loader 0 SHA-1 function asymmetric decrypt function boot0key	section 1 via boot0key
1	boot loader 1 SoPEC_OS_public_key	section 2 via SoPEC_OS_public_key
2	Silverbrook O/S program code function to generate SoPEC_id_key from SoPEC_id Basic Print Engine ComCo_public_key	section 3 via ComCo_public_key section 4 via OEM_public_key (supplied in section 3) PRINTER_QA data, which includes the PrintEngineLicense_id, Silverbrook operating parameters, and OEM operating parameters (all authenticated via SoPEC_id_key)
3	ComCo license agreement operating parameter ranges, including PrintEngineLicense_id (gets loaded into supervisor mode section of memory) OEM_public_key (gets loaded into supervisor mode section of memory) Any ComCo written user-mode program code (gets loaded into mode mode section of memory)	Is used by section 2 to verify section 4 and range of parameters as found in PRINTER_QA
4	OEM specific program code	OEM operating parameters via calls to Silverbrook O/S code

The verification procedures will be required each time the CPU is woken up, since the RAM is not preserved.

3.5.7 What if the CPU is not fast enough?

In the example of Section 3.5.6, every time the CPU is woken up to print a document it needs to perform:

- SHA-1 on all program code and program data
- 4 sets of asymmetric decryption to load the program code and data
- 1 HMAC-SHA1 generation per 512-bits of Silverbrook and OEM printer and ink operating parameters

Although the SHA-1 and HMAC process will be fast enough on the embedded CPU (the program code will be executing from ROM), it may be that the asymmetric decryption will be slow. And this becomes more likely with each extra level of authentication. If this is the case (as is likely), hardware acceleration is required.

A cheap form of hardware acceleration takes advantage of the fact that in most cases the same program is loaded each time, with the first time likely to be at power-up. The hardware acceleration is simply data storage for the *authorizedDigest* which means that the boot procedure now is:

```
slowCPU_bootloader0(data, sig)
   localDigest ← SHA-1(data)
   If (localDigest = previouslyStoredAuthorizedDigest)
      jump to program code at data-start address// will never to return
   Else
      authorizedDigest ← decrypt(sig, boot0key)
      If (localDigest = authorizedDigest)
         previouslyStoredAuthorizedDigest ← authorizedDigest
         jump to program code at data-start address// will never to return
      Else
      // program code is unauthorized
   EndIf
```

This procedure means that a reboot of the same authorized program code will only require SHA-1 processing. At power-up, or if new program code is loaded (e.g. an upgrade of a driver over the internet), then the full authorization via asymmetric decryption takes place. This is because the stored digest will not match at power-up and whenever a new program is loaded.

The question is how much preserved space is required.

Each digest requires 160 bits (20 bytes), and this is constant regardless of the asymmetric encryption scheme or the key length. While it is possible to reduce this number of bits, thereby sacrificing security, the cost is small enough to warrant keeping the full digest.

However each level of boot loader requires its own digest to be preserved. This gives a maximum of 20 bytes per loader. Digests for operating parameters and ink levels may also be preserved in the same way, although these authentications should be fast enough not to require cached storage.

Assuming SoPEC provides for 12 digests (to be generous), this is a total of 240 bytes. These 240 bytes could easily be stored as 60 × 32-bit registers, or probably more conveniently as a small amount of RAM (eg 0.25 - 1 Kbyte). Providing something like 1 Kbyte of RAM has the advantage of allowing the CPU to store other useful data, although this is not a requirement.

In general, it is useful for the boot ROM to know whether it is being started up due to power-on reset or activity on the USB/ISI. In the former case, it can ignore the previously stored values (either 0 for registers or garbage for RAM). In the latter case, it can use the previously stored values. Even without this, a startup value of 0 (or garbage) means the digest won't match and therefore the authentication will occur implictly.

3.6 SoPEC ISI IDENTIFICATION

At power-up, the host can send targeted data to the USB-connected SoPEC, but can only send broadcasts to all of the slave SoPECs via the USB-connected SoPEC's ISI.

Each slave SoPEC will verify the broadcast message received over the ISI, and if it is valid, will execute it. Several levels of authorization may occur. However, at some stage, this common program code (broadcast to all of the slave SoPECs and signed by the appropriate asymmetric private key) will, among other things, set the slave SoPEC's ISI id. If there is only 1 slave, the id is given, but if there is more than 1 slave, the id must be determined in some fashion.

On a particular physical arrangement of SoPECs each slave SoPEC will have a different set of connections on GPIOs. For example, one SoPEC maybe in charge of motor control, while another may be driving the LEDs etc. The unused GPIO pins (not necessarily the same on each SoPEC) can be set as inputs and then tied to 0 or 1. As long as the connection settings are mutually exclusive, program code can determine which is which, and the id appropriately set.

In some multi-SoPEC systems, a given SoPEC will only be attached to a single printhead (left or right). We can conveniently use the second printhead connection pins (temperature and test) to form an ISI id.

This scheme of slave SoPEC identification does not introduce a security breach. If an attacker rewires the pinouts to confuse identification, at best it will simply cause strange printouts (e.g. swapping of printout data) to occur, while at worst the Print Engine will simply not function.

Note that some physical setting (e.g. pins) on each of the multiple SoPECs is required - the settings just need to be mutually exclusive. Although it is possible for all the SoPECs to come to a logical ISI id assignment (e.g. by using ethernet-like protocols), the ISI id needs to be very much a *physical* identity scheme. This is because these SoPECs are not simply logical processors - we want the correct portion of the page to be printed on the correct physical location, motor controls will be physically connected to a specific physical SoPEC etc.

3.7 SETTING UP QA CHIP KEYS

In use, each INK_QA chip needs the following keys:

- K_0 = *SupplyInkLicense_key*
- K_1 = *UseInkLicense_key*

Each PRINTER_QA chip tied to a specific SoPEC requires the following keys:

- K_0 = *PrintEngineLicense_key*
- K_1 = *SoPEC_id_key*
- K_2 = *UseInkLicense_key*

Note that there may be more than one K_1 depending on the number of PRINTER_QA chips and SoPECs in a system. These keys need to be appropriately set up in the QA Chips before they will function correctly together.

3.7.1 Original QA Chips as received by a ComCo

When original QA Chips are shipped from QACo to a specific ComCo their keys are as follows:

- K_0 = *QACo_ComCo_Key0*
- K_1 = *QACo_ComCo_Key1*
- K_2 = *QACo_ComCo_Key2*
- K_3 = *QACo_ComCo_Key3*

All 4 keys are only known to QACo. Note that these keys are different for each QA Chip.

3.7.2 Steps at the ComCo

The ComCo is responsible for making Print Engines out of Memjet printheads, QA Chips, PECs or SoPECs, PCBs etc.

In addition, the ComCo must customize the INK_QA chips and PRINTER_QA chip on-board the print engine before shipping to the OEM.

There are two stages:

- replacing the keys in QA Chips with specific keys for the application (i.e. INK_QA and PRINTER_QA)
- setting operating parameters as per the license with the OEM

3.7.2.1 Replacing keys

The ComCo is issued QID hardware [4] by QACo that allows programming of the various keys (except for K_1) in a given QA Chip to the final values, following the standard ChipF/ChipP replace key (indirect version) protocol [5]. The indirect version of the protocol allows each *QACo_ComCo_Key* to be different for each SoPEC.

In the case of programming of PRINTER_QA's K_1 to be *SoPEC_id_key*, there is the additional step of transferring an asymmetrically encrypted *SoPEC_id_key* (by the public-key) along with the nonce (R_P) used in the replace key protocol to the device that is functioning as a ChipF. The ChipF must decrypt the *SoPEC_id_key* so it can generate the standard replace key message for PRINTER_QA (functioning as a ChipP in the ChipF/ChipP protocol). The asymmetric key pair held in the ChipF equivalent should be unique to a ComCo (but still known only by QACo) to prevent damage in the case of a compromise.

Note that the various keys installed in the QA Chips (both INK_QA and PRINTER_QA) are only known to the QACo. The OEM only uses QIDs and QACo supplied ChipFs. The replace key protocol [5] allows the programming to occur without compromising the old or new key.

3.7.2.2 Setting operating parameters

There are two sets of operating parameters stored in PRINTER_QA and INK_QA:

- fixed
- upgradable

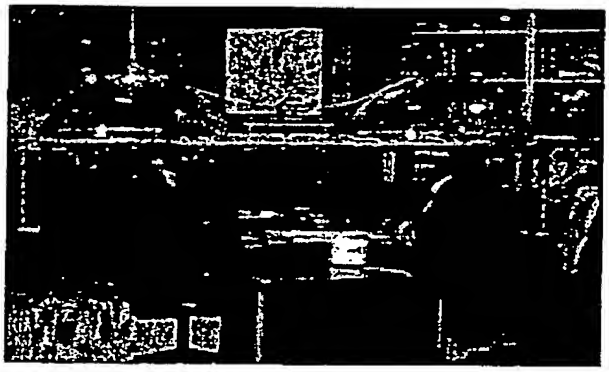
The fixed operating parameters can be written to by means of a non-authenticated writes [5] to M_{1+} via a QID [4], and permission bits set such that they are ReadOnly.

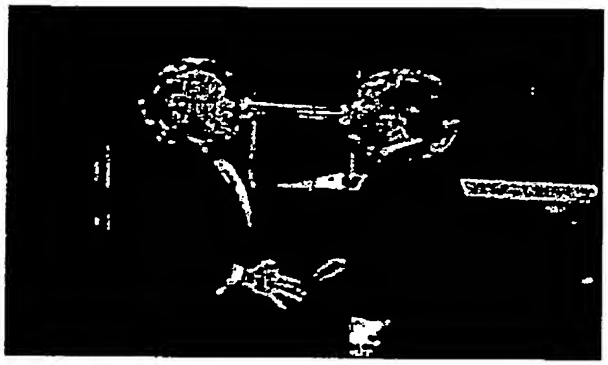
The upgradable operating parameters can only be written to after the QA Chips have been programmed with the correct keys as per Section 3.7.2.1. Once they contain the correct keys they can be programmed with appropriate operating parameters by means of a QID and an appropriate ChipS (containing matching keys).

Authentication Protocols

4-3-1-26 version 0.2

29 November 2002





Silverbrook Research Pty Ltd
393 Darling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9818 6633
Fax: +61 2 9818 6711
Email: info@silverbrookresearch.com

Document History

Version	Date	Authors	Details
0.2	29 November, 2002	Simon Walmsley	Added more detail on programming key stage, and extracted basic form of write, a bit more on batch key programming, and a SoPEC reference.
0.1d	02 October, 2002	Simon Walmsley	initial draft release

Contents

INTRODUCTION

1 Introduction

This document describes authentication protocols for general authentication applications, but with specific reference to the QA Chip [1]. These protocols supersede those described in the Authentications of Consumables document [2].

The intention is to show the broad form of possible protocols for use in different authentication situations, and can be used as a reference when subsequently defining an implementation specification for a particular application.

2 Readership

This document is written for software engineers, hardware engineers, and key management system architects.

This document is confidential to Silverbrook Research Pty. Ltd. and its distribution outside this organisation *must* be covered by a non-disclosure agreement (NDA).

3 References

[1] Silverbrook Research, 2002, *4-3-1-2 QA Chip Technical Reference.*
[2] Silverbrook Research, 1998, *4-3-1-3 Authentication of Consumables.*

4 Nomenclature

The following symbolic nomenclature is used throughout this document:

Table 1. Summary of symbolic nomenclature

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)
X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇩X	Decrement X by 1 (floor 0)
⇧X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z

4.1 PSEUDOCODE

4.1.1 Asynchronous

The following pseudocode:

```
var = expression
```

means the var signal or output is equal to the evaluation of the expression.

4.1.2 Synchronous

The following pseudocode:

```
var ← expression
```

means the var register is assigned the result of evaluating the expression during this cycle.

4.1.3 Expression

Expressions are defined using the nomenclature in Table 1 above. Therefore:

```
var = (a = b)
```

is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

5 Basic Protocols

5.1 PROTOCOL BACKGROUND

This protocol set is a restricted form of a more general case of a multiple key single memory vector protocol. It is a restricted form in that the memory vector M has been optimized for Flash memory utilization:

- M is broken into multiple memory vectors (semi-fixed and variable components) for the purposes of optimizing flash memory utilization. Typically M contains some parts that are fixed at some stage of the manufacturing process (eg a batch number, serial number etc.), and once set, are not ever updated. This information does not contain the amount of consumable remaining, and therefore is not read or written to with any great frequency.
- We therefore define M_0 to be the M that contains the frequently updated sections, and the remaining Ms to be rarely written to. Authenticated writes only write to M_0, and non-authenticated writes can be directed to a specific M_n. This reduces the size of permissions that are stored in the QA Chip (since key-based writes are not required for Ms other than M_0). It also means that M_0 and the remaining Ms can be manipulated in different ways, thereby increasing flash memory longevity.

5.2 REQUIREMENTS OF PROTOCOL

Each QA Chip contains the following values:

N	The maximum number of keys known to the chip.
T	The number of vectors M is broken into.
K_N	Array of *N* secret keys used for calculating $F_{Kn}[X]$ where K_n is the *n*th element of the array.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M_T	Array of T memory vectors. Only M_0 can be written to with an authorized write, while all Ms can be written to in an unauthorized write. Writes to M_0 are optimized for Flash usage, while updates to any other M_{1+} are expensive with regards to Flash utilization, and are expected to be only performed once per section of M_n. M_1 contains T, N and f in ReadOnly form so users of the chip can know these two values.
P_{T+N}	T+N element array of access permissions for each part of M. Entries n={0... T-1} hold access permissions for non-authenticated writes to M_n (no key required). Entries n={T to T+N-1}hold access permissions for authenticated writes to M_0 for K_n. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.
C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a sub-message to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_{Kn}[N,X]$	*Internal function only.* Returns $S_{Kn}[X]$, the result of applying a digital signature function S to X based upon the appropriate key K_n. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme

chosen, although the scheme chosen for the QA Chip is HMAC-SHA1, and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

5.3 READ PROTOCOLS

The set of read protocols describe the means by which a System reads a specific data vector M_t from a QA Chip referred to as *ChipR*.

We assume that the communications link to ChipR (and therefore ChipR itself) is not trusted. If it were trusted, the System could simply read the data and there is no issue. Since the communications link to ChipR is not trusted and ChipR cannot be trusted, the System needs a way of authenticating the data as actually being from a real ChipR.

Since the read protocol must be capable of being implemented in physical QA Chips, we cannot use asymmetric cryptography (for example the ChipR signs the data with a private key, and System validates the signature using a public key).

This document describes two read protocols:

- direct validation of reads
- indirect validation of reads.

5.3.1 Direct Validation of Reads

In a direct validation read protocol we require two QA Chips: *ChipR* is the QA Chip being read, and *ChipT* is the QA Chip we entrust to tell us whether or not the data read from ChipR is trustworthy.

The basic idea is that system asks ChipR for data, and ChipR responds with the data and a signature based on a secret key. System then asks ChipT whether the signature supplied by ChipR is correct. If ChipT responds that it is, then System can trust that data just read from ChipR. Every time data is read from ChipR, the validation procedure must be carried out.

Direct validation requires the System to trust the communication line to ChipT. This could be because ChipT is in physical proximity to the System, and both System and ChipT are in a trusted (e.g. Silverbrook secure) environment. However, since we need to validate the read, ChipR by definition must be in a non-trusted environment.

Each QA Chip protects its signature generation or verification mechanism by the use of a nonce.

The protocol requires the following publicly available functions in ChipT:

Random[] Returns R (does not advance R).

Test[n,X, Y, Z] Advances R and returns 1 if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

The protocol requires the following publicly available functions in ChipR:

Read[n, t, X] Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K. If t is invalid, the function assumes t=0.

To read ChipR's memory M_t in a validated way, System performs the following tasks:

a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipR's Read function, passing in some key number *n1*, the desired data vector number *t*, and R_T (from b);

d. ChipR updates R_R, then calculates and returns R_R, M_{Rt}, $S_{Kn1}[R_T|R_R|C_1|M_{Rt}]$;

e. System calls ChipT's Test function, passing in the key to use for signature verification *n2*, and the results from d (i.e. R_R, M_{Rt}, $S_{Kn1}[R_T|R_R|C_1|M_{Rt}]$);

f. System checks response from ChipT. If the response is 1, then the M_t read from ChipR is considered to be valid. If 0, then the M_t read from ChipR is considered to be invalid.

The choice of n1 and n2 must be such that ChipR's K_{n1} = ChipT's K_{n2}.

The data flow for this read protocol is shown in Figure 1:

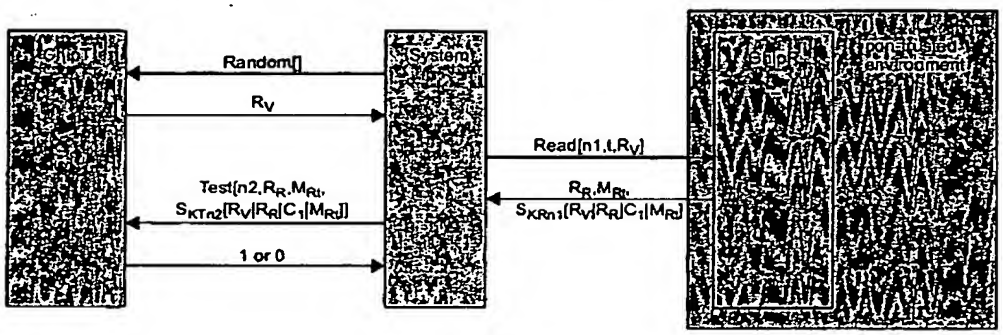


Figure 1. Protocol for directly verifying reads from ChipR

From the System's perspective, the protocol would take on a form like the following pseudocode:

```
R_T ← ChipT.Random()
R_R, M_R, SIG_R ← ChipR.Read(keyNumOnChipR,desiredM, R_T)
ok ← ChipT.Test(keyNumOnChipT, R_R, M_R, SIG_R)
If (ok = 1)
   // M_R is to be trusted
Else
   // M_R is not to be trusted
EndIf
```

With regards to security, if an attacker finds out ChipR's K_{n1}, they can replace the ChipR by a fake ChipR because they can create signatures. Likewise, if an attacker finds out ChipT's K_{n2}, they can replace the ChipR by a fake ChipR because ChipR's K_{n1} = ChipT's K_{n2}. Moreover, they can use the ChipRs on any system that shares the same key.

The only way of restricting exposure due to key reveals is to restrict the number of systems that match ChipR and ChipT. i.e. vary the key as much as possible. The degree to which this can be done will depend on the application. In the case of a PRINTER_QA acting as a ChipT, and an INK_QA acting as a ChipR, the same key must be used on all systems where the particular INK_QA data must be validated.

In all cases, ChipR must contain sufficient information to produce a signature. Knowing (or finding out) this information, whatever form it is in, allows clone ChipRs to be built.

5.3.2 Indirect Validation of Reads

In a direct validation protocol (see Section 5.3.1), the System validates the correctness of data read from ChipR by means of a trusted chip ChipT. This is possible because ChipR and ChipT share some secret information.

However, it is possible to extend trust via indirect validation. This is required when we trust ChipT, but ChipT doesn't know how to validate data from ChipR. Instead, ChipT knows how to validate data from *ChipI* (some intermediate chip) which in turn knows how to validate data from either another ChipI (and so on up a chain) or ChipR. Thus we have a chain of validation.

The means of validation chains is translation of signatures. $ChipI_n$ translates signatures from higher up the chain (either $ChipI_{n-1}$ or from ChipR at the start of the chain) into signatures capable of being passed to the next stage in the chain (either $ChipI_{n+1}$ or to ChipT at the end of the chain). A given ChipI can only translate signatures if it knows the key of the previous stage in the chain as well as the key of the next stage in the chain.

The protocol requires the following publicly available functions in ChipI:

Random[] Returns R (does not advance R).

Translate[n1,X, Y, Z,n2,A] Returns 1, $S_{Kn2}[A|R|C_1|Y]$ and advances R if Z = $S_{Kn1}[R|X|C_1|Y]$. Otherwise returns 0, 0. The time taken to calculate and compare signatures must be independent of data content.

The data flow for this signature translation protocol is shown in Figure 2:

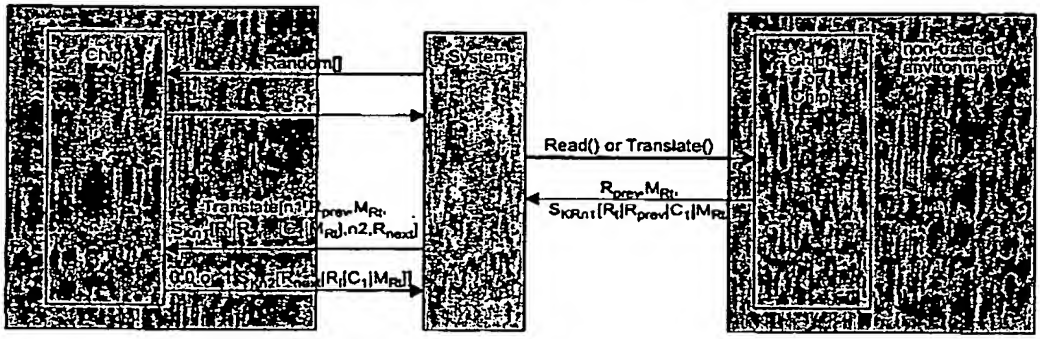


Figure 2. Protocol for signature translation protocol

Note that R_{prev} is eventually R_R, and R_{next} is eventually R_T. In the multiple ChipI case, R_{prev} is the R_I of $ChipI_{n-1}$ and R_{next} is R_I of $ChipI_{n+1}$. The R_{prev} of the first ChipI in the chain is R_R, and the R_{next} of the last ChipI in the chain is R_T.

Assuming at least 1 ChipT, the System would need to perform the following tasks in order to read ChipR's memory M_t in an indirectly validated way:

a. System calls $ChipI_n$'s Random function;

b. $ChipI_0$ returns R_{I0} to System;

c. System calls ChipR's Read function, passing in some key number *n0*, the desired data vector number *t*, and R_{I0} (from b);

d. ChipR updates R_R, then calculates and returns R_R, M_{Rt}, $S_{Kn0}[R_{In}|R_R|C_I|M_{Rt}]$;

e. System assigns R_R to R_{prev} and $S_{Kn0}[R_{In}|R_R|C_I|M_{Rt}]$ to SIG_{prev}

f. System calls the next-chip-in-the-chain's Random function (either $ChipI_{n+1}$ or ChipT)

g. The next-chip-in-the-chain will return R_{next} to System

h. System calls $ChipI_n$'s Translate function, passing in $n1_n$ (translation input key number), R_{prev}, M_{Rt}, SIG_{prev}), $n2_n$ (translation output key number) and the results from g (R_{next});

i. ChipI returns testResult and SIG_I to System

j. If testResult = 0, then the validation has failed, and the M_t read from ChipR is considered to be invalid. Exit with failure.

k. If the next chip in the chain is a ChipI, assign SIG_I to SIG_{prev} and go to step f

l. System calls ChipT's Test function, passing in n_t, R_{prev}, M_{Rt}, and SIG_{prev};

m. System calls System checks response from ChipT. If the response is 1, then the M_t read from ChipR is considered to be valid. If 0, then the M_t read from ChipR is considered to be invalid.

For the Translate function to work, $ChipI_n$ and $ChipI_{n+1}$ must share a key. The choice of n1 and n2 in the protocol described must be such that $ChipI_n$'s K_{n2} = $ChipI_{n+1}$'s K_{n1}.

Note that Translate is essentially a "Test plus resign" function. From an implementation point of view the first part of Translate is identical to Test.

Note that the use of ChipIs and the translate function merely allows signatures to be transformed. At the end of the translation chain (if present) will be a ChipT requiring the use of a Test function. There can be any number of ChipIs in the chain to ChipT as long as the Translate function is used to map signatures between $ChipI_n$ and $ChipI_{n+1}$ and so on until arrival at the final destination (ChipT).

From the System's perspective, a read protocol using at least 1 ChipI would take on a form like the following pseudocode:

```
R_next ← ChipI[0].Random()
R_prev, M_R, SIG_prev ← ChipR.Read(keyNumOnChipR,desiredM, R_next)
ok = 1
i = 0
while ((i < iMax) AND ok)
For i ← 0 to iMax
   If (i = iMax)
      R_next ← ChipT.Random()
   Else
      R_next ← ChipI[i+1].Random()
   EndIf
   ok, SIG_prev ← ChipI[i].Translate(iKey[i], R_prev, M_R, SIG_prev, oKey[i], R_next)
   R_prev = R_next
   If (ok = 0)
      // M_R is not to be trusted
   EndIf
EndFor
ok ← ChipT.Test(keyNumOnChipT, R_prev, M_R, SIG_prev)
If (ok = 1)
   // M_R is to be trusted
```

```
Else
   // M_R is not to be trusted
EndIf
```

5.3.3 Additional Comments on Reads

In the Memjet printing environment, certain implementations will exist where the operating parameters are stored in QA Chips. In this case, the system must read the data from the QA Chip using an appropriate read protocol.

If the connection is trusted (e.g. to a virtual QA Chip in software), a generic Read is sufficient. If the connection is not trusted, it is ideal that the System have a trusted ChipT in the form of software (if possible) or hardware (e.g. a QA Chip on board the same silicon package as the microcontroller and firmware). Whether implemented in software or hardware, the QA Chip should contain an appropriate key that is unique per print engine. Such a key setup would allow reads of print engine parameters and also allow indirect reads of consumables (from a consumable QA Chip).

If the ChipT is physically separate from System (e.g. ChipT is on a board connected to System) System ***must also occasionally*** (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to reduce the possibility of someone inserting a fake ChipT into the system that always returns 1 for the Test function.

5.4 UPGRADE PROTOCOLS

This set of protocols describe the means by which a System upgrades a specific data vector M_t within a QA Chip (*ChipU*). The data vector may contain information about the functioning of the device (e.g. the current maximum operating speed) or the amount of a consumable remaining.

The updating of M_t in ChipU falls into two categories:

- non-authenticated writes, where anyone is able to update the data vector
- authenticated writes, where only authorized entities are able to upgrades data vectors

5.4.1 Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation for M_0, and during the manufacturing process for M_{1+}.

In this kind of write, the System wants to change M_t within ChipU subject to P. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know and of the Ks or even have access to a trusted chip* to perform the write, the System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires ChipU to contain the following publicly available function:

Write[t, X] Writes X over those parts of M_t subject to P_t and the existing value for M.

To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};

b. The authentication procedure for a Read is carried out (see Section 5.3 on page 5);

c. If the read succeeds in such a way that M_{new} = M returned in **b**, the write succeeded. If not, it failed.

Note that if these parameters are transmitted over an error-prone communications line (as opposed to internally or using an additional error-free transport layer), then an additional checksum would be required to prevent the wrong M from being updated or to prevent the correct M from being updated to the wrong value. For example, SHA-1[t,X] should be additionally transferred across the communications line and checked (either by a wrapper function around Write or in a variant of Write that takes a hash as an extra parameter).

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation for M_0, and during the manufacturing process for M_{1+}.

5.4.2 Authenticated writes

In the QA Chip protocols, M_0 is defined to be the only data vector that can be upgraded in an authenticated way. This decision was made primarily to simplify flash management, although it also helps to reduce the permissions storage requirements.

In this kind of write, System wants to change Chip U's M_0 in an authorized way, without being subject to the permissions that apply during normal operation. For example, a consumable may be at a refilling station and the normally Decrement Only section of M_0 should be updated to include the new valid consumable. In this case, the chip whose M_0 is being updated must authenticate the writes being generated by the external System and in addition, apply the appropriate permission for the key to ensure that only the correct parts of M_0 are updated. Having a different permission for each key is required as when multiple keys are involved, all keys should not necessarily be given open access to M_0. For example, suppose M_0 contains printer speed and a counter of money available for franking. A ChipS that updates printer speed should not be capable of updating the amount of money. Since $P_{0...T-1}$ is used for non-authenticated writes, each K_n has a corresponding permission P_{T+n} that determines what can be updated in an authenticated write.

The basic principle of the authenticated write (or upgrade) protocol is that the new value for the M_t must be signed before ChipU accepts it. The QA Chip responsible for generating the signature (ChipS) must first validate that the ChipU is valid by reading the old value for M_t. Once the old value is seen as valid, a new value can be signed by ChipS and the resultant data plus signature passed to ChipU. Note that both chips distrust each other.

There are two forms of authenticated writes. The first form is when both ChipU and ChipS directly store the same key. The second is when both ChipU and ChipS store different versions of the key and a transforming procedure is used on the stored key to generate the required key - i.e. the key is indirectly stored. The second form is slightly more complicated, and only has value when the ChipS is not readily available to an attacker.

5.4.2.1 direct authenticated writes

The direct form of the authenticated write protocol is used when the ChipS and ChipU are equally available to an attacker. For example, suppose that ChipU contains a printer's operating speed. Suppose that the speed can be increased by purchasing a ChipS and inserting it into the printer system. In this case, the ChipS and ChipU are equally available to an attacker. This is different from upgrading the printer over the internet where the effective ChipS is in a remote location, and thereby not as readily available to an attacker.

The direct authenticated write protocol requires ChipU to contain the following publicly available functions:

Read[n, t, X] Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K.

WriteA[n, X, Y, Z] Advances R, replaces M_0 by Y subject to P_{T+n}, and returns 1 only if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y subject to P_{T+n} to its M when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures.

In its basic form, ChipS requires the following variables and function:

SignM[n,V,W,X,Y,Z] Advances R, and returns R, $S_{Kn}[W|R|C_1|Z]$ only if $Y = S_{Kn}[V|W|C_1|X]$. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1, 0 (desired vector number) and 0 (the random value, but is a don't-care value) as the input parameters;

b. ChipU produces R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (the random value as used in **a**), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S and $S_{Kn2}[R_U|R_S|C_1|M_D]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in **d** are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from **d**.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes is shown in Figure 3:

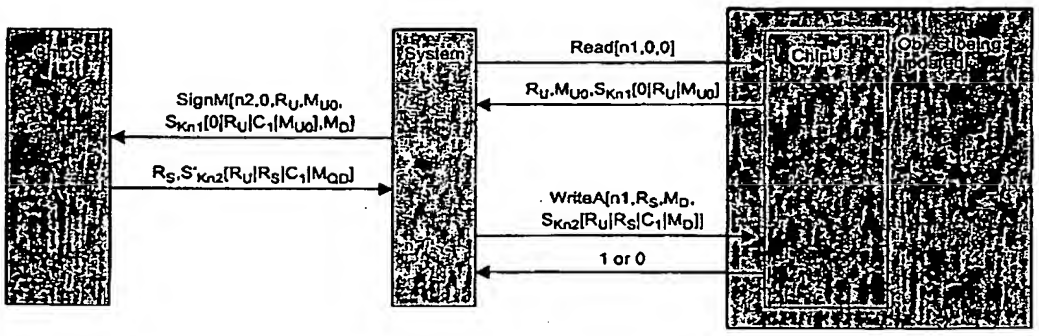


Figure 3. Protocol for direct authenticated write

Note that this protocol allows ChipS to generate a signature for any desired memory vector MD, and therefore a stolen ChipS has the ability to effectively render the particular keys for those parts of M_0 in ChipU irrelevant.

It is therefore not recommended that the basic form of ChipS be ever implemented except in specifically controlled circumstances.

It is much more secure to limit the powers of ChipS. The following list covers some of the variants of limiting the power of ChipS:

a. the ability to upgrade a limited number of times

b. the ability to upgrade based on a credit value - i.e. the upgrade amount is decremented from the local value, and effectively transferred to the upgraded device

c. the ability to upgrade to a fixed value or from a limited list

d. the ability to upgrade to any value

e. the ability to only upgrade certain data fields within M

In many of these variants, the ability to refresh the ChipS in some way (e.g. with a new count or credit value) would be a useful feature.

In certain cases, the variant is in ChipS, while ChipU remains the same. It may also be desirable to create a ChipU variant, for example only allowing ChipU to only be upgraded a specific number of times.

5.4.2.1.1 variant example

This section details the variant for the ability to upgrade a memory vector to any value a specific number of times, but the upgrade is only allowed to affect certain fields within the memory vector i.e. a combination of (a), (d), and (e) above.

In this example, ChipS requires the following variables and function:

CountRemaining Part of ChipS's M_0 that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M_0 needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M_0 (assuming ChipS's Ps allows that part of M_0 to be updated).

Q Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's $P_{0..T-1}$ for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[n,V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{Kn}[W|R|C_1|Z_{QX}]$ only if $Y = S_{Kn}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1, 0 (desired vector number) and 0 (the random value, but is a don't-care value) as the input parameters;

b. ChipU produces R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_{U0} using Q) and $S_{Kn2}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from d.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for this variant of authenticated writes is shown in Figure 4:

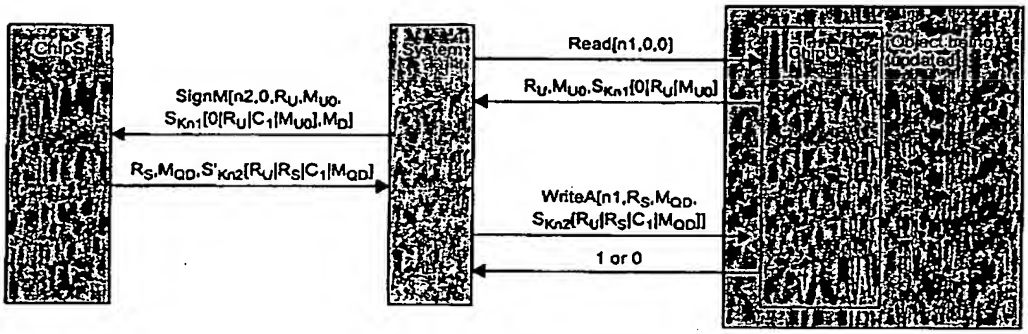


Figure 4. Protocol for direct authenticated write - variant

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by $P_{0..T-1}$ set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

In addition, CountRemaining in ChipS needs to be setup (including making it ReadOnly in P_S) before ChipS is programmed with K_U. ChipS should therefore be programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

5.4.2.2 *Indirect authenticated writes*

This section describes an alternative authenticated write protocol when ChipU is more readily available to an attacker and ChipS is less available to an attacker. We can store different keys on ChipU and ChipS, and implement a mapping between them in such a way that if the attacker is able to obtain a key from a given ChipU, they cannot upgrade all ChipUs.

In the general case, this is accomplished by storing key K_S on ChipS, and K_U and f on ChipU. The relationship is $f(K_S) = K_U$ such that knowledge of K_U and f does not make it easy to determine K_S. This implies that a one-way function is desirable for f.

In the QA Chip domain, we define f as a number (e.g. 32-bits) such that SHA1(K_S | f) = K_U. The value of f (random between chips) can be stored in a known location within M_1 as a constant for the life of the QA Chip. It is possible to use the same f for multiple relationships if desired, since f is public and the protection lies in the fact that f varies between QA Chips (preferably in a non-predictable way).

The indirect protocol is the same as the direct protocol with the exception that f is additionally passed in to the SignM function so that ChipS is able to generate the correct key. The System obtains f by performing a Read of M_1. Note that all other functions, including the WriteA function in ChipU, are identical to their direct authentication counterparts.

SignM[f,n,V,W,X,Y,Z] Advances R, and returns R, $S_{f(Kn)}[W|R|C_1|Z]$ only if Y = $S_{f(Kn)}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

Before reading ChipU's memory M_0 (the pre-upgrade value), the System must extract f from ChipU by performing the following tasks:

a. System calls ChipU's Read function, passing in (dontCare, 1, dontCare)

b. ChipU returns M_1, from which System can extract f_U

c. System stores f_U for future use

To update ChipU's M vector, the protocol is identical to that described in the basic authenticated write protocol with the exception of steps c and d:

c. System calls ChipS's SignM function, passing in f_U, n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S and $S_{fU(Kn2)}[R_U|R_S|C_1|M_D]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

In addition, the choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's $f_U(K_{n2})$.

Note that f_U is obtained from M_1 *without validation*. This is because there is nothing to be gained by subverting the value of f_U, (because then the signatures won't match).

From the System's perspective, the protocol would take on a form like the following pseudocode:

```
dontCare, M_R, dontCare ← ChipR.Read(dontCare,1, dontCare)
f_R = extract from M_R
...
R_U, M_U, SIG_U ← ChipU.Read(keyNumOnChipU,0, 0)
R_S, SIG_S = ChipS.SignM2(f_R, keyNumOnChipS, 0, R_U, M_U, SIG_U, M_D)
If (R_S = SIG_S = 0)
   // ChipU and therefore M_U is not to be trusted
Else
   // ChipU and therefore M_U can be trusted
   ok = ChipU.WriteA(keyNumOnChipU, R_S, M_D, SIG_S)
   If (ok)
      // updating of data in ChipU was successful
   Else
      // transmission error during WriteA
   EndIf
EndIf
```

5.4.2.2.1 variant example

The indirect form of the example from Section 5.4.2.1.1 is shown here.

SignM[f,n,V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{f(Kn)}[W|R|C_1|Z_{QX}]$ only if Y = $S_{f(Kn)}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

Before reading ChipU's memory M_0 (the pre-upgrade value), the System must extract f from ChipU by performing the following tasks:

a. System calls ChipU's Read function, passing in (dontCare, 1, dontCare)

b. ChipU returns M_1, from which System can extract f_U

c. System stores f_U for future use

To update ChipU's M vector, the protocol is identical to that described in the basic authenticated write protocol with the exception of steps c and d:

c. System calls ChipS's SignM function, passing in f_U, n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_{U0} using Q) and $S_{fU(Kn2)}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

In addition, the choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's $f_U(K_{n2})$.

Note that f_U is obtained from M_1 *without validation*. This is because there is nothing to be gained by subverting the value of f_U, (because then the signatures won't match).

From the System's perspective, the protocol would take on a form like the following pseudocode:

```
dontCare, MR, dontCare ← ChipR.Read(dontCare,1, dontCare)
fR = extract from MR
...
RU, MU, SIGU ← ChipU.Read(keyNumOnChipU,0, 0)
RS, MQD, SIGS = ChipS.SignM2(fR, keyNumOnChipS, 0, RU, MU, SIGU, MD)
If (RS = MQD = SIGS = 0)
   // ChipU and therefore MU is not to be trusted
Else
   // ChipU and therefore MU can be trusted
   ok = ChipU.WriteA(keyNumOnChipU, RS, MQD, SIGS)
   If (ok)
      // updating of data in ChipU was successful
   Else
      // transmission error during WriteA
   EndIf
EndIf
```

5.4.3 Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P (and certain parts of M), only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all M. As different processes fill up different parts of M, they can be

sealed against future change by updating the permissions. Updating a chip's $P_{0..T-1}$ changes permissions for unauthorized writes to M_n, and updating $P_{T..T+N-1}$ changes permissions for authorized writes with key K_n.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Random[] Returns R (does not advance R).

SetPermission[n,p,X,Y,Z] Advances R, and updates P_p according to Y and returns 1 followed by the resultant P_p only if $S_{Kn}[R|X|Y|C_2] = Z$. Otherwise returns 0. P_p can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_p).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variable:

CountRemaining Part of ChipS's M_0 that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M_0 needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M_0 (assuming ChipS's P_n allows that part of M_0 to be updated).

In addition, ChipS requires either of the following two SignP functions depending on whether direct or indirect key storage is used (see direct vs indirect authenticated write protocols in Section 5.4.2):

SignP[n,X,Y] Used when the same key is directly stored in both ChipS and ChipU. Advances R, decrements CountRemaining and returns R and $S_{Kn}[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

SignP[f,n,X,Y] Used when the same key is not directly stored in both ChipS and ChipU. In this case ChipU's K_{n1} = ChipS's $f(K_{n2})$. The function is identical to the direct form of SignP, except that it additionally accepts f and returns $S_{f(Kn)}[X|R|Y|C_2]$ instead of $S_{Kn}[X|R|Y|C_2]$.

5.4.3.1 *Direct form of SignP*

When the direct form of SignP is used, ChipU's P_n is updated as follows:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in n2, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{Kn2}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with n1, the desired permission entry p, R_S, P_D and $S_{Kn2}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against its own generated signature $S_{Kn1}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for basic authenticated writes to permissions is shown in Figure 5:

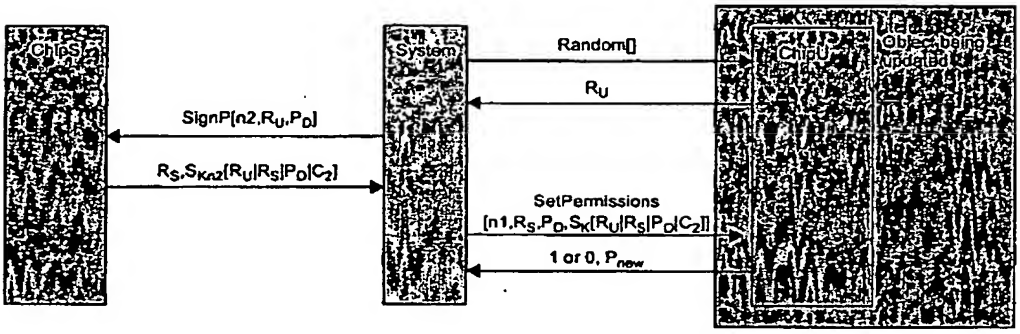


Figure 5. Protocol for basic update of permissions

5.4.3.2 *Indirect form of SignP*

When the indirect form of SignP is used in ChipS, the System must extract f from ChipU (so it knows how to generate the correct key) by performing the following tasks:

a. System calls ChipU's Read function, passing in (dontCare, 1, dontCare)

b. ChipU returns M_1, from which System can extract f_U

c. System stores f_U for future use

ChipU's P_n is updated as follows:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in f_U, n2, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{fU(Kn2)}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with n1, the desired permission entry p, R_S, P_D and $S_{fU(Kn2)}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_{Kn1}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

In addition, the choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's $f_U(K_{n2})$.

5.4.4 Protecting memory vectors

To protect the appropriate part of M_n against unauthorized writes, call SetPermissions[n] for n = 0 to T-1. To protect the appropriate part of M_0 against authorized writes with key n, call SetPermissions[T+n] for n=0 to N-1.

Note that only M_0 can be written in an authenticated fashion.

Note that the SetPermission function must be called *after* the part of M has been set to the desired value.

For example, if adding a serial number to an area of M_1 that is currently ReadWrite so that noone is permitted to update the number again:

- the Write function is called to write the serial number to M_1
- SetPermission(1) is called for to set that part of M to be ReadOnly for non-authorized writes.

If adding a consumable value to M_0 such that only keys 1-2 can update it, and keys 0, and 3-N cannot:

- the Write function is called to write the amount of consumable to M
- SetPermission is called for 0 to set that part of M_0 to be DecrementOnly for *non-authorized* writes. This allows the amount of consumable to decrement.
- SetPermission is called for n = {T, T+3, T+4 ..., T+N-1} to set that part of M_0 to be ReadOnly for *authorized* writes using all but keys 1 and 2. This leaves keys 1 and 2 with ReadWrite permissions to M_0.

It is possible for someone who knows a key to further restrict other keys, but it is not in anyone's interest to do so.

5.5 PROGRAMMING K

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

The protocol assumes that ChipF and ChipP already share (directly or indirectly) a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

Although the example shows a ChipF that is only allowed to program a specific number of ChipPs, the key-upgrade protocol can be easily altered (similar to the way the write protocols have variants) to provide other means of limiting the ability to update ChipPs.

The protocol requires the following publicly available functions in ChipP:

Random[] Returns R (does not advance R).

ReplaceKey[n, X, Y, Z] Replaces K_n by $S_{Kn}[R|X|C_3] \oplus Y$, advances R, and returns 1 only if $S_{Kn}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

And the following data and functions in ChipF:

CountRemaining Part of M_0 with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M_0 needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M_0.

K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible. After manufacture, K_{new} is 0.

SetPartialKey[X] Updates K_{new} to be $K_{new} \oplus X$. This function allows K_{new} to be programmed in any number of steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory.

In addition, ChipF requires either of the following GetProgramKey functions depending on whether direct or indirect key storage is used on the input key and/or output key (see direct vs indirect authenticated write protocols in Section 5.4.2):

GetProgramKey1[n, X] Direct to direct. Used when the same key (K_n) is directly stored in both ChipF and ChipP and we want to store K_{new} in ChipP. Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kn}[X|R_F|C_3] \oplus K_{new}$ and a signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

GetProgramKey2[f, n, X] Direct to indirect. Used when the same key (K_n) is directly stored in both ChipF and ChipP but we want to store $f_P(K_{new})$ in ChipP instead of simply K_{new} (i.e. we want to keep the key in ChipP to be different in all ChipPs). In this case ChipP's K_{n1} = ChipF's $f_P(K_{n2})$. The function is identical to GetProgramKey1, except that it additionally accepts f_P and returns $S_{Kn}[X|R_F|C_3] \oplus f_P(K_{new})$ instead of $S_{Kn}[X|R_F|C_3] \oplus K_{new}$. Note that the produced signature is produced using K_n since that is what is already stored in ChipP.

GetProgramKey3[f, n, X] Indirect to direct. Used when the same key is not directly stored in both ChipF and ChipP but we want to store K_{new} in ChipP. In this case ChipP's K_{n1} = ChipF's $f_P(K_{n2})$. The function is identical to GetProgramKey1, except that it additionally accepts f_P and returns $S_{fP(Kn)}[X|R_F|C_3] \oplus K_{new}$ instead of $S_{Kn}[X|R_F|C_3] \oplus K_{new}$. The produced signature is produced using $f_P(Kn)$ instead of K_n since that is what is already stored in ChipP.

GetProgramKey4[f, n, X] Indirect to indirect. Used when the same key is not directly stored in both ChipF and ChipP but we want to store $f_P(K_{new})$ in ChipP instead of simply K_{new} (i.e. we want to keep the key in ChipP to be different in all ChipPs). In this case ChipP's K_{n1} = ChipF's $f_P(K_{n2})$. The function is identical to GetProgramKey3, except that it returns $S_{fP(Kn)}[X|R_F|C_3] \oplus f_P(K_{new})$ instead of $S_{fP(Kn)}[X|R_F|C_3] \oplus K_{new}$. The produced signature is produced using $f_P(K_n)$ since that is what is already stored in ChipP.

Since there are likely to be few ChipFs, and many ChipPs, the indirect forms of GetProgramKey can be usefully employed.

5.5.1 GetProgramKey1 - direct to direct

With the "old key = direct, new key = direct" form of GetProgramKey, to update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in n2 (the desired key to use) and the result from b;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn2}[R_P|R_F|C_3] \oplus K_{new}$, and $S_{Kn2}[R_F|S_{Kn2}[R_P|R_F|C_3] \oplus K_{new}|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from **d**;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to K_{new}. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's K_{n2}.

The data flow for key updates is shown in Figure 6:

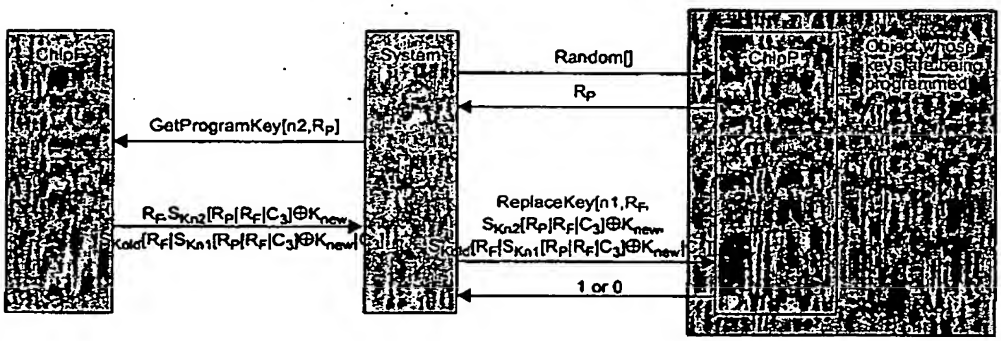


Figure 6. Protocol for multiple key update

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kn2}[R_P|R_F|C_3]$ without K_{n2}. The signature based on K_{new} is sent to ensure that ChipP will be able to determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_{F0} (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if necessary (see Section 5.4.2 on page 10).

5.5.2 GetProgramKey2 - direct to indirect

With the "old key = direct, new key = indirect" form of GetProgramKey, to update P's key, the System must extract f from ChipP (so it can tell ChipF how to generate the correct key) by performing the following tasks:

a. System calls ChipP's Read function, passing in (dontCare, 1, dontCare)

b. ChipP returns M_1, from which System can extract f_P

c. System stores f_P for future use

ChipP's key is updated as follows:

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in f_P, n2 (the desired key to use) and the result from **b**;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn2}[R_P|R_F|C_3]\oplus f_P(K_{new})$, and $S_{Kn2}[R_F|S_{Kn2}[R_P|R_F|C_3]\oplus f_P(K_{new})|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from **d**;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to $f_P(K_{new})$. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's K_{n2}.

5.5.3 GetProgramKey3 - indirect to direct

With the "old key = indirect, new key = direct" form of GetProgramKey, to update P's key, the System must extract f from ChipP (so it can tell ChipF how to generate the correct key) by performing the following tasks:

a. System calls ChipP's Read function, passing in (dontCare, 1, dontCare)

b. ChipP returns M_1, from which System can extract f_P

c. System stores f_P for future use

ChipP's key is updated as follows:

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in f_P, n2 (the desired key to use) and the result from **b**;

d. ChipF updates R_F, then calculates and returns R_F, $S_{fP(Kn2)}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{fP(Kn2)}[R_F|S_{fP(Kn2)}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from **d**;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to K_{new}. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's $f_P(K_{n2})$.

5.5.4 GetProgramKey4 - indirect to indirect

With the "old key = indirect, new key = indirect" form of GetProgramKey, to update P's key, the System must extract f from ChipP (so it can tell ChipF how to generate the correct key) by performing the following tasks:

a. System calls ChipP's Read function, passing in (dontCare, 1, dontCare)

b. ChipP returns M_1, from which System can extract f_P

c. System stores f_P for future use

ChipP's key is updated as follows:

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in f_P, n2 (the desired key to use) and the result from **b**;

d. ChipF updates R_F, then calculates and returns R_F, $S_{fP(Kn2)}[R_P|R_F|C_3]\oplus f_P(K_{new})$, and $S_{fP(Kn2)}[R_F|S_{fP(Kn2)}[R_P|R_F|C_3]\oplus f_P(K_{new})|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in n1 (the key to use in ChipP) and the response from **d**;

f. System checks response from ChipP. If the response is 1, then ChipP's K_{n1} has been correctly updated to $f_P(K_{new})$. If the response is 0, ChipP's K_{n1} has not been updated.

The choice of n1 and n2 must be such that ChipP's K_{n1} = ChipF's $f_P(K_{n2})$.

5.5.5 Chicken and Egg

The Program Key protocol requires both ChipF and ChipP to know K_{old} (either directly or indirectly). Obviously both chips had to be programmed in some way with K_{old}, and thus K_{old} can be thought of as an older K_{new}. K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc., and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks. This is especially true if K_{first} is indirectly stored in ChipPs (i.e. each ChipP holds an f and $f(K_{first})$ instead of K_{first} directly). One example is where K_{first} (the key stored in each chip after manufacture/test) is a batch key, and can be different per chip. K_{first} may advance to a ComCo specific K_{second} etc. but still remain indirect. A direct form (e.g. K_{final}) only needs to go in if it is actually required at the end of the programming chain.

Depending on reprogramming requirements, K_{first} can be the same or different for all K_n.

5.5.6 Security Note

It is imperative that different ChipFs have different R_F values to prevent K_{new} from being determined as follows:

The attacker needs 2 ChipFs, both with the same R_F and K_n but different values for K_{new}. By knowing K_{new1} the attacker can determine K_{new2}. The size of R_F is 2^{160}, and assuming a lifespan of approximately 2^{32} Rs, an attacker needs about 2^{60} ChipFs with the same K_n to locate the correct chip. Given that there are likely to be only hundreds of ChipFs with the same K_n, this is not a likely attack. The attack can be eliminated completely by making C_3 different per chip and transmitting it with the new signature.

6 Memjet forms of Protocols

Physical QA Chips are used in Memjet printer systems to store printer operating parameters as well as consumable parameters.

6.1 PRINTER_QA

A PRINTER_QA is stored within each print engine to perform two primary tasks:

- storage and protection of operating parameters
- a means of indirect read validation of other QA Chip data vectors

Each PRINTER_QA contains the following keys:

Table 2. Keys in PrinterQA

Key	Contents	Comments
0	Upgrade Key	Used to upgrade the operating parameters. Should be indirect form of key (i.e. a different key for each PRINTER_QA) so that an indirect form of the write is required.
1	Consumable Read Validation Key	Used to indirectly read the data from an CONSUMABLE_QA chip using indirect authenticated read protocol (Section 5.3.2 on page 7).
2	PrintEngineController Read Validation Key	When reading data from the PRINTER_QA, the system can either trust the data, or must use this key to perform the authenticated read protocol (see Section 5.3 on page 5).
3-n	(reserved)	Currently unused. Could be used to provide a means to indirectly read additional print engine operating parameters ala K1, or provide additional Print Engine validation ala K2.

Note that if multiple Print Engine Controllers are used (e.g. a multiple SoPEC system), then multiple PrintEngineController Read Validation Keys are required. These keys can be stored within a single PRINTER_QA (e.g. in K_3 and beyond), or can be stored in separate PRINTER_QAs (for example each SoPEC (or group of SoPECs) has an individual PRINTER_QA).

The functions required in the PRINTER_QA are:

- **Random, ReplaceKey**, to allow key programming & substitution
- **Read**, to allow reads of data
- **Write**, to allow updates of M_{1+} during manufacture
- **WriteAuth**, to provide a means of updating the M_0 data (operating parameters)
- **SetPermissions**, to provide a means of updating write permissions
- **Test**, to provide a means of checking if consumable reads are valid
- **Translate**, to provide a means of indirect reading of consumable data

6.2 CONSUMABLE_QA

A CONSUMABLE_QA is stored with each consumable (e.g. ink cartridge) to perform two primary tasks:

- storage of consumable related data
- protection of consumable amount remaining

Each CONSUMABLE_QA contains the following keys:

Table 3. Keys in CONSUMABLE_QA

Key	Contents	Comments
0	Upgrade Key	Used to upgrade the consumable parameters. Should be stored as the indirect form of the key (i.e. a different key for each CONSUMABLE_QA) so that an indirect form of the write is required.
1	Consumable Read Validation Key	When reading data from the CONSUMABLE_QA, the system can either trust the data, or must use this key to perform either the direct or indirect authenticated read protocol (see Section 5.3 on page 5).
2	(reserved)	Currently unused.
3-n	(reserved)	Currently unused.

The functions required in the CONSUMABLE_QA are:

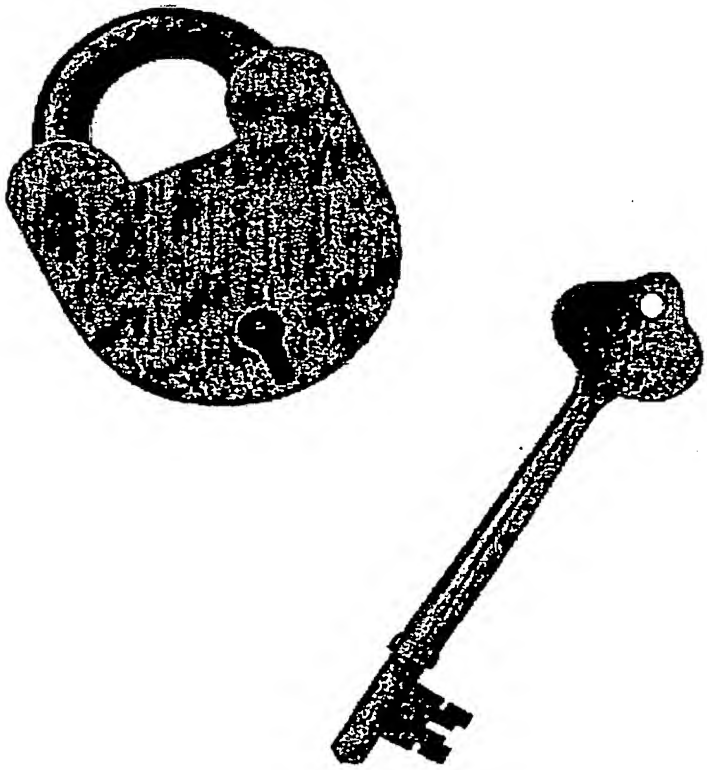
- **Random, ReplaceKey,** to allow key programming & substitution
- **Read,** to allow reads of data
- **Write,** to allow updates of M_{1+} during manufacture
- **WriteAuth,** to provide a means of updating the M_0 data (consumable remaining)
- **SetPermissions,** to provide a means of updating write permissions

QA Chip

Technical Reference

4-3-1-2 version 4.01

29 November 2002



Silverbrook Research Pty Ltd
393 Darling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9818 6633
Fax: +61 2 9818 6711
Email: info@silverbrookresearch.com

Confidential

Document History

Version	Date	Authors	Details
4.01	29 November, 2002	Simon Walmsley	PMU: fixed problem in PMU where the fuse was being read at startup (when IOMode = idle). Fuse is now only read when entering Program Mode, and the mask is invalidated when in Trim or Program Mode (thus re-reading correctly when entering Active Mode). Fixed problem in MRU - not running code in flash when unshdowed
4.0	25 November, 2002	Simon Walmsley Gary Shipton (MAU)	Stopped MRU from ever erasing memory automatically - instead the retries are recorded for reading back under CPU control, and a facility that hangs the current CPU program added instead of erasing memory. The fact that the retries saturated can be recovered after program restart. Removed PMFuseIntact from going into the MRU - can have non-shadowing when the fuse is blown. Added deglitching to IOU. Added MAU detail from Gary. Extracted Authentication of Consumables out to 4-3-1-3 so that this document represents implementation of the chip.
3.10d	11 November, 2002	Simon Walmsley	Added retry facility to bad reads in MRU, up to 15 times, then erases the chip if was to info memory.
3.9d	7 November, 2002	Simon Walmsley	Small typos & fixes to 3.8d.

Version	Date	Authors	Details
3.8d	28 October, 2002	Simon Walmsley	IOU: Removal of the OutByteValidZ 2-bit register, and addition of a single AckCmd bit. The AckCmd bit is set in the byte state machine checkId state when EndByte occurs. It is used in the bit state machine's TxAck and makes the TxAck considerably simpler. PCU: Stack is now 12 entries deep instead of 8 entries. PCU, AGU: The PCRamSel register added to the PCU - set when RST, JSI or JPI instruction (and fetch). This is now transmitted and used by the AGU to set accessMode for PC accesses instead of always using 0 (read flash only). PMU: Writes to main, reads from info memory since fuse & chipMask are now in info memory MRU: updated to allow flash testing and for information memory access by PMU. Extensive changes to permit flash testing. Changes include 8-bit non-shadowed accesses, writes to flash test registers in both the MRU and MAU, selection of info block and main block in low order 128 bytes (non shadowed, 64 bytes shadowed) etc. MAU: has now a programmable timer for duration of writes and erasures when selected. Also provides write access to the 5-bit test regiter within the TSMC flash IP.
3.7d	23 September, 2002	Simon Walmsley	MRU: Added mechanism to allow substitution of bad nybbles instead of erasure PMU/FEU: fuse & seal mechanism updated to be based on value in flash instead of an electically programmable bit.

Version	Date	Authors	Details
3.6d	21 August, 2002	Simon Walmsley	MRU: * incorporated Mask Unit into MRU so that erased flash is processed on a per nybble basis. Extensive rewrite of MRU. * erased memory is detected as all 1s, but is transformed into all 0s to reduce program code space in CPU. IOU: Fixed problem of receipt of a byte when a InByte had not yet been read (previously ACKed) - now does not load the new byte, returns to idle state and hence does not ACK (therefore NAKs). *Note this does not cause a return to IOMode = Idle.* Fixed power-down mode in IOU to work with 3 stop bits properly. Added text to describe 2 D-types on the inputs of SDa and SClk. Removed DIE opcode, and removed the effects of this from the FEU and AGU (the CPU now only outputs 5 bits for AccessMode instead of 6). DIE should instead be implemented by writing to location 0 and a RST instruction, thereby causing a shadow error which will reoccur next bootup should the power be removed before the erasure has started.
3.5d	25 July, 2002	Simon Walmsley	MRU reset state & calculation of rw, fixed calculation of allowTransaction when doing flash erasures. FEU: fixed reset state of NewMemTrans, and generation of fetch, resetGo & seal.
3.4d	15 July, 2002	Simon Walmsley	Split RWAdr from MIU into FlashAdr and RAMAdr to make testing easier.
3.3d	5 July, 2002	Simon Walmsley	Defined byte values for Trim Mode, Active Mode, Program Mode, and globalId. Added further text for Trim Mode. Changed MRU: * default for non-action is read rather than write * masked flash writes no longer produce flash reads. Instead they produce no MAU requests (MRUNewTrans = 0)
3.2d	1 July, 2002	Gary Shipton	Added trim mode explanation.
3.1d	28 June, 2002	Simon Walmsley	Added immediate addressing, reduced address registers to 4. Added JMP, JPI. Deleted CLR and SET. Increased PC stack depth to 8. Added trim mode. Changed FEU to allow pendingStart and pendingKill.

Version	Date	Author	Details
3.0d	8 April, 2002	Simon Walmsley	Added multiple keys and multiple memory vectors. Improved protocols. Added shadow flash. All addressing is now relative.
2.07d	25 September, 2001	Simon Walmsley	Changes due to feedback on manufacturing test vectors - increase the testability of the device: * deleted MTRZ * added ability to read MTR register * inverted sense of MTR within MTU * added extra constants (0, 1, FF, and FFFFFFFF) to save constant space * added constant 0xFFFFFFFF as operand to ADD and XOR to allow effective DEC and NOT instructions * added ability to read Timer register
2.06d	1 August, 2001	Simon Walmsley	Fixed TxBit (in IOU) to be synchronous.
2.05	27 July, 2001	Simon Walmsley	Changes to allow authenticated writes: * added 16 more words of RAM * removed IST, ISW, AM, and MT * removed CHK from direct addressing * reduced JSI to 8 addresses only (3 bits) * shifted PK, DBR, TBR and program code start addresses due to JSI change * renamed K2MX to K2SEL since it no longer has any bearing on M * added MYR 1-bit flag to select between Flash R/M and RAM R/M, including SET MYR and CLR MYR instructions * updated SC C2 instruction to shift in counter values * shifted addresses of M, and K * only change in MIU is in MTRU, where MTRU is updated due to size of RAM and change of K address in Flash * RPL HI & LO updated to work on appropriate part of Acc * deleted BitGen and RPLGen from AGU * updated JSIGen, OFFGen, VarGen due to new addressing requirements * reduced input to rw register in AGU since BitGen is no longer required * moved ROR operands around a bit to add ROR 8 instruction * X[N4] operand moved to be the same for all opcodes. Is therefore now available to AND and OR opcodes.
2.04	4 July, 2001	Simon Walmsley	Bug fixes based on software simulation and FPGA implementation.

Version	Date	Authors	Details
2.03	23 April 2001	Simon Walmsley	Typo and bug fixes. Added Tamper Detection Unit plus implementation of how reset and erasure tamper detection outputs are used.
2.02	9 April 2001	Simon Walmsley	Removal of BIST. Fetch & Exec are now asynchronous. Abstraction of blocks into CPU and MIU. Detail added to MIU components.
2.01	19 March 2001	Simon Walmsley	Small typo fixes.
2.0	19 March 2001	Simon Walmsley	Extensive revamp of entire chip implementation: Separation of scannable from non-scannable sections, included BIST, defined BIST Mode, Program Mode, Active Mode, Idle Mode, changes in MU, AGU, FE, IOU, MTU, changed the way constants are handled, added opcodes for accessing GO bit, Atomic instruction wrapping provided (disables interrupts), new LDK operation mechanism. Global ID vs LocalID authentication chip commands. Increased program memory size to 768 bytes. Merged variable flash memory and program memory into 1 flash region, changed the way erasures work, allow multiple cycle 0 and cycle 1 with Fetch & Exec signals set during the last cycle of Cycle 0 and Cycle 1 respectively, address generation changed to match separation of memory blocks (memory mapped regions of memory removed).
1.4	14 February, 2001	Simon Walmsley	Changed serial bit operations to byte based operations (update to instruction set, IO unit and ALU). Deleted Program Mode Detection Unit. Updated text in IO unit to describe the serial I/O protocol, and briefly described program mode.
1.3	23 December, 1998	Simon Walmsley	Converted to FrameMaker. Updated after review of theory document. Added CHK, FMN, FMH, FML. Increased program memory size to 512 bytes. Added Checking of K1 and K2.
1.2	30 July, 1998	Simon Walmsley	Fixed LFSR taps in error. Added NOP. Changed TBR to work via offsets (like DBR). Fixed SP wrapping. Swapped bit-patterns for TBR, DBR and JSR so startup is via JSR instead of TBR.
1.1	1 July, 1998	Simon Walmsley	Split into 2 documents: This one (chip implementation), and theory.
1.0	5 June, 1998	Simon Walmsley	Initial issue.

Contents

INTRODUCTION

1 Background

Manufacturers of systems that require consumables (such as a laser printer that requires toner cartridges) have struggled with the problem of authenticating consumables, to varying levels of success. Most have resorted to specialized packaging that involves a patent. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. The prevention of copying is important for two reasons:

- To protect revenues
- To prevent poorly manufactured substitute consumables from damaging the base system. For example, poorly filtered ink may clog print nozzles in an ink jet printer, causing the consumer to blame the system manufacturer and not admit the use of non-authorized consumables.

This document describes a QA Chip that can be used to hold contains authentication keys together with circuitry specially designed to prevent copying. The chip is manufactured using a standard Flash memory manufacturing process, and is low cost enough to be included in consumables such as ink and toner cartridges. The implementation is approximately $1mm^2$ in a 0.25 micron flash process, and has an expected die manufacturing cost of approximately 10 cents in 2003.

Once programmed, the QA Chips as described here are compliant with the NSA export guidelines since they do not constitute a strong encryption device. They can therefore be practically manufactured in the USA (and exported) or anywhere else in the world.

Note that although the QA Chip is designed for use in authentication systems, it is microcoded, and can therefore be programmed for a variety of applications.

2 Readership

This document is written for hardware engineers, software engineers, production engineers and system architects.

This document is confidential to Silverbrook Research Pty. Ltd. and its distribution outside this organisation *must* be covered by a non-disclosure agreement (NDA).

3 Associated Documents

[1] describes the problems of consumable authentication, while [2] describes authentication protocols specifically designed to solve problems identified in [1].

[3] describes the implementation of the QA Chip Interface Device (QID), used for testing the QA Chips after manufacture, and used to program each QA Chip with its application specific microcode.

4 Nomenclature

The following symbolic nomenclature is used throughout this document:

Table 1. Summary of symbolic nomenclature

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)
X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇩X	Decrement X by 1 (floor 0)
⇧X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z

4.1 PSEUDOCODE

4.1.1 Asynchronous

The following pseudocode:

```
var = expression
```

means the var signal or output is equal to the evaluation of the expression.

4.1.2 Synchronous

The following pseudocode:

```
var ← expression
```

means the var register is assigned the result of evaluating the expression during this cycle.

4.1.3 Expression

Expressions are defined using the nomenclature in Table 1 above. Therefore:

```
var = (a = b)
```

is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

4.2 DIAGRAMS

Black lines are used to denote data, while red lines are used to denote 1-bit control-signal lines.

LOGICAL INTERFACE

5 Introduction

The QA Chip has a physical and a logical external interface. The physical interface defines how the QA Chip can be connected to a physical System, while the logical interface determines how that System can communicate with the QA Chip. This section deals with the logical interface.

5.1 Operating Modes

The QA Chip has four operating modes - *Idle Mode*, *Program Mode*, *Trim Mode* and *Active Mode*.

- *Idle Mode* is used to allow the chip to wait for the next instruction from the System.
- *Trim Mode* is used to determine the clock speed of the chip and to trim the frequency during the initial programming stage of the chip (when Flash memory is garbage). The clock frequency *must* be trimmed via Trim Mode *before* Program Mode is used to store the program code.
- *Program Mode* is used to load up the operating program code, and is required because the operating program code is stored in Flash memory instead of ROM (for security reasons).
- *Active Mode* is used to execute the specific authentication command specified by the System. Program code is executed in *Active Mode*. When the results of the command have been returned to the System, the chip enters *Idle Mode* to wait for the next instruction.

5.1.1 Idle Mode

The QA Chip starts up in *Idle Mode*. When the Chip is in *Idle Mode*, it waits for a command from the master by watching the low speed serial line for an id that matches either the global id (0x00), or the chip's local id.

- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Trim Mode id byte, the QA Chip enters *Trim Mode* and starts counting the number of internal clock cycles until the next byte is received.
- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Program Mode id byte, the QA Chip enters *Program Mode*.
- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Active Mode id byte, the QA Chip enters *Active Mode* and executes startup code, allowing the chip to set itself into a state to receive authentication commands (includes setting a local id).
- If the primary id matches the chip's local id, and the following byte is a valid command code, the QA Chip enters *Active Mode*, allowing the command to be executed.

The valid 8-bit serial mode values sent after a global id are as shown in Table 2. They are specified to minimize the chances of them occurring by error after a global id (e.g. 0xFF and 0x00 are not used):

Table 2. Id byte values to place chip in specific mode

Value	Interpretation
10100101 (0xA5)	Trim Mode
10001110 (0x8E)	Program Mode
01111000 (0x78)	Active Mode

5.1.2 Trim Mode

Trim Mode is enabled by sending a global id byte (0x00) followed by the Trim Mode command byte.

The purpose of Trim Mode is to set the trim value (an internal register setting) of the internal ring oscillator so that Flash erasures and writes are of the correct duration. This is necessary due to the variation of the clock speed due to process variations. If writes an erasures are too long, the Flash memory will wear out faster than desired, and in some cases can even be damaged.

Trim Mode works by measuring the number of system clock cycles that occur inside the chip from the receipt of the Trim Mode command byte until the receipt of a data byte. When the data byte is received, the data byte is copied to the trim register and the current value of the count is transmitted to the outside world.

Once the count has been transmitted, the QA Chip returns to *Idle Mode*.

At reset, the internal trim register setting is set to a known value r. The external user can now perform the following operations:

- send the global id+write followed by the Trim Mode command byte
- send the 8-bit value v over a specified time t
- send a stop bit to signify no more data
- send the global id+read followed by the Trim Mode command byte
- receive the count c
- send a stop bit to signify no more data

At the end of this procedure, the trim register will be v, and the external user will know the relationship between external time t and internal time c. Therefore a new value for v can be calculated.

The Trim Mode procedure can be repeated a number of times, varying both t and v in known ways, measuring the resultant c. At the end of the process, the final value for v is established (and stored in the trim register for subsequent use in Program Mode). This value v must also be written to the flash for later use (every time the chip is placed in Active Mode for the first time after power-up).

For more information about the internal workings of Trim Mode and the accuracy of trim in the QA Chip, see Section 11.2 on page 42.

5.1.3 Program Mode

Program Mode is enabled by sending a global id byte (0x00) followed by the Program Mode command byte.

The QA Chip determines whether or not the internal fuse has been blown (by reading 32-bit word 0 of the information block of flash memory).

If the fuse has been blown the Program Mode command is ignored, and the QA Chip returns to *Idle Mode*.

If the fuse is still intact, the chip enters Program Mode and erases the entire contents of Flash memory. The QA Chip then validates the erasure. If the erasure was successful, the

QA Chip receives up to 4096 bytes of data corresponding to the new program code and variable data. The bytes are transferred in order $byte_0$ to $byte_{4095}$.

Once all bytes of data have been loaded into Flash, the QA Chip returns to *Idle Mode*.

Note that Trim Mode functionality *must be performed before* a chip enters Program Mode for the first time.

Once the desired number of bytes have been downloaded in Program Mode, the LSS Master must wait for 80µs (the time taken to write two bytes to flash at nybble rates) before sending the new transaction (eg Active Mode). Otherwise the last nybbles may not be written to flash.

5.1.4 Active Mode

Active Mode is entered either by receiving a global id byte (0x00) followed by the Active Mode command byte, or by sending a local id byte followed by a command opcode byte and an appropriate number of data bytes representing the required input parameters for that opcode.

In both cases, Active Mode causes execution of program code previously stored in the flash memory via Program Mode. As a result, we never enter Active Mode after Trim Mode, without a Program Mode in between. However once programmed via Program Mode, a chip is allowed to enter Active Mode after power-up, since valid data will be in flash.

If Active Mode is entered by the global id mechanism, the QA Chip executes specific reset startup code, typically setting up the local id and other IO specific data.

If Active Mode is entered by the local id mechanism, the QA Chip executes specific code depending on the following byte, which functions as an opcode. The opcode command byte format is shown in Table 3. The transmission of the opcode with its inverse assists in the detection of transmission errors:

Table 3. Command byte

bits	description
2-0	opcode
5-3	¬opcode
7-6	count of number of bits set in opcode (0 to 3)

The interpretation of the 3-bit opcode depends on whatever software happens to be stored in the QA Chip.

LOGICAL VIEW OF CPU

6 Introduction

Authentication logic is implemented via microcode executing in a special purpose CPU, using instructions and memory models tailor-made for this purpose. The high level commands that a user of an QA Chip sees are all implemented as small programs written in the CPU instruction set.

The following sections describe the memory model, the various registers, and the instruction set of the CPU.

7 Memory Model

The QA Chip has its own internal memory, broken into the following conceptual regions:

- ***RAM variables*** (3Kbits = 96 entries at 32-bits wide), used scratch storage (e.g. HMAC-SHA1 processing).
- ***Flash memory*** (8Kbytes main block + 128 bytes info block) used to hold the non-volatile authentication variables (including program keys etc), and program code. Only 4 KBytes + 64 bytes is visible to the program addressing space due to shadowing. Shadowing is where half of each byte is used to validate and verify the other half, thus protecting against certain forms of physical and logical attacks. As a result, two bytes are read to obtain a single byte of data (this happens transparently).

7.1 RAM

The RAM region consists of 96 × 32-bit words required for the general functioning of the QA Chip, *but only during the operation of the chip*. RAM is volatile memory: once power is removed, the values are lost. Note that in actual fact memory retains its value for some period of time after power-down, but cannot be considered to be available upon power-up. This has issues for security that are addressed in other sections of this document.

RAM is used for temporary storage of variables during chip operation. Short programs can also be stored and executed from the RAM.

RAM is addressed from 0 to 5F. Since RAM is in an unknown state upon a RESET (RstL), program code should not assume the contents to be 0.

7.2 FLASH VARIABLES

The flash memory region contains the non-volatile information in the QA Chip. Flash memory retains its value after a RESET or if power is removed, and can be expected to be unchanged when the power is next turned on.

Byte 0 of main memory is the first byte of the program run for the command dispatcher. Note that the command dispatcher is always run with shadows enabled.

Bytes 0-7 of the information block flash memory is reserved as follows:

- byte 0-3 = fuse. A value of 0x5555AAAA indicates that the fuse is no longer intact (think of a physical fuse whose wire is intact or not)
- bytes 4-7 = random number used to XOR all data for RAM and flash memory accesses

After Program Mode or a globalId Active command, the 32-bit data from bytes 4-7 in the information block of Flash memory is loaded into an internal ChipMask register. In Active Mode (the chip is executing program code), all data read from the flash and RAM is XORed with the ChipMask register, and all data written to the flash and RAM is XORed with the ChipMask register before being written out. This XORing happens completely transparently to the program code. Main flash memory byte 0 onward is the start of program code. Note that byte 0 onward needs to be valid after being XORed with the appropriate bytes of ChipMask.

Multi-word variables are most likely to be accessed conveniently if they are are stored most significant word first due to addressing requirements. The addressing scheme used is a base address (given by an address register) offset by an index that starts at some number

n and decrements to 0. Thus (An),n is the first word accessed, and (An),0 is the last 32-bit word accessed in loop processing.

Even though CPU access is in 8-bit and 32-bit quantities, the data is actually stored in flash a nybble-at-a-time. Each nybble write is written as a byte containing 4 sets of b/¬b pairs. Thus every byte write to flash is writing a nybble to real and shadow. A WriteMask allows the individual targetting of nybble-at-a-time writes.

The checking of flash vs shadow flash is automatically carried out each read (each byte contains both flash and shadow flash). If all 8 bits are 1, the byte is considered to be in its erased form[1], and returns 0 as the nybble. Otherwise, the value returned for the nybble depends on the size of the overall access and the setting of bit 0 of the WriteMask.

- All 8-bit accesses (i.e. instruction and program code fetches) are checked to ensure that each byte read from flash is 4 sets of b/¬b pairs. If the data is not of this form, the chip hangs until a new command is issued over the serial interface.
- With 32-bit accesses (i.e. data used by program code), each byte read from flash is checked to ensure that it is 4 sets of b/¬b pairs. A setting of $WriteMask_0 = 0$ means that if the data is not valid, then the chip will hang until a new command is issued over the serial interface. A setting of $WriteMask_0 = 1$ means that each invalid nybble is replaced by the upper nybble of the WriteMask. This allows recovery after a write or erasure is interrupted by a power-down.

1. TSMC's flash memory has an erased state of all 1s

8 Registers

A number of registers are defined for use by the CPU. They are used for control, temporary storage, arithmetic functions, counting and indexing, and for I/O.

These registers do not need to be kept in non-volatile (Flash) memory. They can be read or written without the need for an erase cycle (unlike Flash memory). Temporary storage registers that contain secret information still need to be protected from physical attack by Tamper Prevention and Detection circuitry and parity checks.

All registers are cleared to 0 on a RESET, with the exception of WriteMask which is set to all 1s. However, program code should not assume any RAM contents have any particular state, and should set up register values appropriately. In particular, at the startup entry point, the various address registers need to be set up from unknown states.

8.1 GO

A 1-bit GO register is 1 when the program is executing, and 0 when it is not. Programs can clear the GO register to halt execution of program code once the authentication command has finished executing.

8.2 ACCUMULATOR AND Z FLAG

The Accumulator is a 32-bit general-purpose register. It is used as one of the inputs to all arithmetic operations, and is the register used for transferring information between memory registers.

The Z register is a 1-bit flag, and is updated each time the Accumulator is written to. The Z register contains the zero-ness of the Accumulator. Z = 1 if the last value written to the Accumulator was 0, and 0 if the last value written was non-0.

Both the Accumulator and Z registers are directly accessible from the instruction set.

8.3 ADDRESS REGISTERS

8.3.1 Program Counter Array and Stack Pointer

A 12-level deep 12-bit Program Counter Array (PCA) is defined. It is indexed by a 4-bit Stack Pointer (SP). The current Program Counter (PC), containing the address of the currently executing instruction, is effectively PCA[SP].

The PC is affected by calling subroutines or returning from them, and by executing branching instructions. The SP is affected by calling subroutines or returning from them. There is no bounds checking on calling too many subroutines: the oldest entry in the execution stack will be lost.

The entry point for program code is defined to be the program address stored at entry 0 in the JSR Table. This entry point is used whenever the master signals a new transaction.

8.3.2 A0-A3

There are 4 8-bit address registers Each register has an associated memory mode bit designating the address as in Flash (0) or RAM (1).

When an A*n* register is pointing to an address in RAM, it holds the word number. When it is pointing to an address in Flash, it points to a set of 32-bit words that start at a 128-bit (16 byte) alignment.

The A0 register has a special use of direct offset e.g. access is possible to (A0),0-7 which is the 32-bit word pointed to by A0 offset by the specified number of words.

8.3.3 WriteMask

The WriteMask register is used to determine how many nybbles will be written during a 32-bit write to Flash, and whether or not an invalid nybble will be replaced during a read from Flash.

During writes to flash, bit *n* (of 8) determines whether nybble *n* is written. The unit of writing is a nybble since half of each byte is used for shadow data. A setting of 0xFF means that all 32-bits will be written to flash (as 8 sets of nybble writes).

During 32-bit reads from flash (occurs as 8 reads), the value of $WriteMask_0$ is used to determine whether a read of invalid data is replaced by the upper nybble of WriteMask. If 0, a read of invalid data *is not* replaced, and the chip hangs until a new command is issued over the serial interface. If 1, a read of invalid data *is* replaced by the upper nybble of the WriteMask.

Thus a WriteMask setting of 0 (reset setting) means that no writes will occur to flash, and all reads are not replaced (causing the program to hang if an invalid value is encountered).

8.4 COUNTERS

A number of special purpose counters/index registers are defined:

Table 4. Counter/Index registers

Name	Register Size	Bits	Description
C1	1 × 3	3	Counter used to index arrays and general purpose counter
C2	1 × 6	6	General purpose counter and can be used to index arrays

All these counter registers are directly accessible from the instruction set. Special instructions exist to load them with specific values, and other instructions exist to decrement or increment them, or to branch depending on the whether or not the specific counter is zero.

There are also 2 special flags (not registers) associated with C1 and C2, and these flags hold the zero-ness of C1 or C2. The flags are used for loop control, and are listed here, for although they are not registers, they can be tested like registers.

Table 5. Flags for testing C1 and C2

Name	Description
C1Z	1 = C1 is current zero, 0 = C1 is currently non-zero.
C2Z	1 = C2 is current zero, 0 = C2 is currently non-zero.

8.5 RTMP

The single bit register RTMP exists to allow the implementation of LFSRs and multiple precision shift registers.

During a rotate right (ROR) instruction with operand of RB, the bit shifted out (formally bit 0) is written to the RTMP register. The bit currently in the RTMP register becomes the new bit 31 of the Accumulator. Performing multiple ROR RB commands over several 32-bit values implements a multiple precision rotate/shift right.

The XRB operand operates in the same way as RB, in that the current value in the RTMP register becomes the new bit 31 of the Accumulator. However with the XRB instruction, the bit formally known as bit 0 does not simply replace RTMP (as in the RB instruction). Instead, it is XORed with RTMP, and the result stored in RTMP. This allows the implementation of long LFSRs, as required by the authentication protocol.

8.6 REGISTERS USED FOR I/O

Several registers are defined for communication between the master and the QA Chip. These registers are LocalId, InByte and OutByte.

LocalId (6 bits) defines the chip-specific id that this particular QA Chip will accept commands for. InByte (8 bits) provides the means for the QA Chip to obtain the next byte from the master. OutByte (8 bits) provides the means for the QA Chip to send a byte of data to the master.

From the QA Chip's point of view:

- Reads from InByte will hang until there is 1 byte of data present from the master.
- Writes to OutByte will hang if the master has not already consumed the last OutByte.

When the master begins a new authentication command transaction, any existing data in InByte and OutByte is lost, and the PC is reset to the entry point in the code, thus ensuring correct framing of data.

8.7 REGISTERS USED FOR TRIMMING CLOCK SPEED

A single 8-bit Trim register is used to trim the ring oscillaor clock speed. The register has a known value of 0x80 during reset, and can be set in one of two ways:

- via Trim Mode, which is necessary before the QA Chip is programmed for the first time; or
- via the CPU, which is necessary every time the QA Chip is powered up before any flash write or erasure accesses can be carried out

8.8 REGISTERS USED FOR TESTING FLASH

There are a number of registers specifically for testing the flash implementation. A single 32-bit write to an appropriate RAM address allows the setting of any combination of these flash test registers.

RAM consists of 96 × 32-bit words, and can be pointed to by any of the standard A*n* address registers. A write to a RAM address in the range 97-127 does nothing with the RAM (reads return 0), but a write to a RAM address in the range 0x80-0x87 will write to specific groupings of registers according to the low 3 bits of the RAM address. A 1 in the address bit means the appropriate part of the 32-bit Accumulator value will be written to the appropriate flash test registers. A 0 in the address bit means the register bits will be unaffected.

The registers and address bit groupings are listed in Table 6:

Table 6. Flash test registers settable from CPU in RAM address range 0x80-0x87[a]

address bit	data bits	name	description
0	0	shadowsOff	0 = shadowing applies (nybble based flash access) 1 = shadowing disabled, 8-bit direct accesses to flash.
	1	hiFlashAdr	Only valid when shadowsOff = 1 0 = accesses are to lower 4Kbytes of flash 1 = accesses are to upper 4 Kbytes of flash
	2	infoBlockSel	0 = 32-bit accesses to lower 128[b] / 64[c] bytes of flash is to main memory 1 = accesses are to information block
1	3	enableFlashTest	0 = keep flash test register within the TSMC flash IP in its reset state 1 = enable flash test register to take on non-reset values.
	8-4	flashTest	Internal 5-bit flash test register within the TSMC flash IP (SFC008_08B9_HE). If this is written with 0x1E, then subsequent writes will be according to the TSMC write test mode. You must write a non-0x1E value or reset the register to exit this mode.
2	28-9	flashTime	When timerSel is 1, this value is used for the duration of the program cycle within a standard flash write or erasure. 1 unit = 16 clock cycles (16 × 100ns typical). Regardless of timerSel, this value is also used for the timeout following power down detection before the QA Chip resets itself. 1 unit = 1 clock cycle (= 100ns typical). *Note that this means the programmer should set this to an appropriate value (e.g. 5 µs), just as the localId needs to be set.*
	29	timerSel	0 = use internal (default) timings for flash writes & erasures 1 = use flashTime for flash writes and erasures

a. This is from the programmer's perspective. Addresses sent from the CPU are byte aligned, so the MRU needs to test bit n+2. Similarly, checking DRAM address > 128 means testing bit 7 of the address in the CPU, and bit 9 in the MRU.
b. unshadowed
c. shadowed

When none of the address register bits 0-2 are set (e.g. a write to RAM address 0x80), then invalid writes will clear the illChip and retryCount registers.

For example, set the A0 register to be 0x80 in RAM. A write to (A0),0 will write to none of the flash test registers, but will clear the illChip and retryCount registers. A write to (A0),7 will write to all of the flash test registers. A write to (A0),2 will write to the enableFlashTest and flashTest registers only. A write to (A0),4 will write to the flashTime and timerSel registers etc.

Finally, a write to address 0x88 in RAM will cause a device erasure. If infoBlockSel is 0, then the device erasure will only be of main memory. If infoBlockSel is 1, then the device erasure is of both main memory and the information block (which will also clear the ChipMask and the Fuse).

Reads of invalid RAM areas will reveal information as follows:

- all invalid addresses in RAM (e.g. 0x80) will return the illChip flag in the low bit (illChip is set whenever 16 consecutive bad reads occur for a single byte in memory)
- all invalid addresses in RAM with the low address bit set (e.g. 0x81, or (A0),1 when A0 holds 0x80), will additionally return the most recent retryCount setting (only updated by the chip when a bad read occurs). i.e. bit 0 = illChip, bits 4-1 = retryCount.

8.9 REGISTER SUMMARY

Table 7 provides a summary of the registers used in the CPU.

Table 7. Register summary

Register name	Description	#bits
A[0-3]	address registers	4x9 =36
Acc	accumulator	32
C1	general purpose counter and index	3
C2	general purpose counter and index	6
illChip	gets set whenever more than 15 consecutive bad reads from flash occurred (and any program executing has hung)	1
InByte	input byte from outside world	8
Go	determines whether CPU is executing	1
LocalId	determines id for this chip's IO	6
OutByte	output byte to outside world	8
Z	zero flag for last xfer to Acc	1
PCA	program counter array	12x12=144
PCRamSel	Program code is executing in flash (0) or ram (1)	1
RetryCount	counts the number of retries for bad reads	4
RTMP	bit used to alow multi-word rotations	1
SP	stack pointer into PCA	4
Trim	trims ring oscillator frequency	8
flash test registers	various registers in the embedded flash and flash access logic specifically for testing the flash memory	30
	TOTAL (bits)	294

8.10 STARTUP

Whenever the chip is started up, the PC is set to 0, which means execution begins at 0, which will typically be the command dispatcher. The command dispatcher can conveniently set the localID and the Trim register.

9 Instruction Set

The CPU operates on 8-bit instructions and typically on 32-bit data items. Each instruction typically consists of an opcode and operand, although the number of bits allocated to opcode and operand varies between instructions.

9.1 Basic Opcodes (Summary)

The opcodes are summarized in Table 8:

Table 8. Opcode bit pattern map

Opcode	Mnemonic	Simple Description
0000xxxx	JMP	Jump
0001xxxx	JSR	Jump subroutine
0010xxxx	TBR	Test and branch
0011xxxx	DBR	Decrement and branch
0100xxxx	SC	Set counter to a value
0101xxxx	ST	Store Accumulator in specified location
0110000x	-	reserved
01100010	RST	Reset
01100011	JPI	Jump indirect
011001xx	-	reserved
01101xxx	-	reserved
01110000	-	reserved
01110001	ERA	Erase page of flash memory pointed to by Accumulator
01110010	-	reserved
01110011	JSI	Jump subroutine indirect
01110100	RTS	Return from subroutine
01110101	HALT	Stop the CPU
0111011x	-	reserved
01111xxx	LIA	Load immediate value into address register
10000xxx	AND	Bitwise AND Accumulator
10001xxx	OR	Bitwise OR Accumulator
1001xxxx	XOR	Exclusive-OR Accumulator
1010xxxx	ADD	Add a 32 bit value to the Accumulator
1011xxxx	LD	Load Accumulator
1100xxxx	ROR	Rotate Accumulator right
11010xxx	AND	Bitwise AND Accumulator[a]
11011xxx	OR	Bitwise OR Accumulator[a]
11100xxx	XOR	Bitwise XOR Accumulator[a]
11101xxx	ADD	Add a 32 bit value to the Accumulator[a]
11110xxx	LD	Load Accumulator[a]
11111xxx	RIA	Rotate Accumulator into address register

a. Immediate form of instruction

Table 9 is a summary of valid operands for each opcode. The table is ordered alphabetically by opcode mnemonic. The binary value for each operand can be found in the subsequent sections.

Table 9. Valid operands for opcodes

Opcode	Valid Operands
ADD	immediate value (A0), offset (An), {C1,C2} [where n = 0-3]
AND	immediate value (A0), offset
DBR	{C1, C2}, offset
ERA	
HALT	
JMP	address
JPI	
JSI	
JSR	address
LIA	{Flash,Ram}, An [where n = 0-3], {immediate value}
LD	immediate value (A0), offset (An), {C1,C2} [where n = 0-3]
OR	immediate value (A0), offset
RIA	{Flash, Ram}, An [where n = 0-3]
ROR	{InByte, OutByte, WriteMask, ID, C1, C2, RB, XRB, 1,3,8,24,31}
RST	
RTS	
SC	{C1, C2}, {immediate value}
ST	(A0), offset (An), {C1,C2} [where n = 0-3]
TBR	{0, 1}, offset
XOR	immediate value (A0), offset (An), {C1,C2} [where n = 0-3]

Additional pseduo-opcodes (for programming convenience) are as follows:

- DEC=ADD 0xFF..
- INC= ADD 0x01
- NOT=XOR 0xFF..
- LDZ = LD 0
- SC {C1, C2}, Acc = ROR {C1, C2}
- RD = ROR Inbyte
- WR = ROR OutByte
- LDMASK = ROR WriteMask
- LDID = ROR Id
- NOP = XOR 0

9.2 ADDRESSING MODES

The CPU supports a set of addressing modes as follows:

- immediate
- accumulator indirect
- indirect fixed
- indirect indexed

9.2.1 Immediate

In this form of addressing, the operand itself supplies the 32-bit data.

Immediate addressing relies on 3 bits of operand, plus an optional 8 bits at PC+1 to determine an 8-bit base value. Bits 0 to 1 of the opcode byte determine whether the base value comes from the opcode byte itself, or from PC+1, as shown in Table 10.

Table 10. Selection for base value in immediate mode

$Opcode_{1-0}$	BaseValue
00	00000000
01	00000001
10	From PC+1 (i.e. $MIUData_{7-0}$)
11	11111111

The base value is computed by using CMD_0 as bit 0, and copying CMD_1 into the upper 7 bits.

The resultant 8 bit base value is then used as a 32-bit value, with 0s in the upper 24 bits, or the 8-bit value is replicated into the upper 32 bits. The selection is determined by bit 2 of the opcode byte, as follows:

Table 11. Replicate bits selection

$Opcode_2$	Data
0	No replication. Data has 0 in upper 24 bits and baseVal in lower 8 bits
1	Replicated. Data is 32-bit value formed by replicating baseVal.

Opcodes that support immediate addressing are LD, ADD, XOR, AND, OR. The SC and LIA instructions are also immediate in that they store the data with the opcode, but they are not in the same form as that described here. See the detail on the individual instructions for more information.

Single byte examples include:

- LD 0
- ADD 1
- ADD 0xFF... # this subtracts 1 from the acc
- XOR 0xFF... # this performs an effective logical NOT operation

Double byte examples include:

- LD 0x05 # a constant
- AND 0x0F # isolates the lower nybble
- LD 0x36... # useful for HMAC processing

9.2.2 Accumulator indirect

In this form of addressing, the Accumulator holds the effective address.

Opcodes that support Accumulator indirect addressing are JPI, JSI and ERA. In the case of JPI and JSI, the Accumulator holds the address to jump to. In the case of ERA, the Accumulator holds the address of the page in flash memory to be erased.

Examples include:

- JPI
- JSI
- ERA

9.2.3 Indirect fixed

In this form of addressing, address register A0 is used as a base address, and then a specific fixed offset is added to the base address to give the effective address.

Bits 2-0 of the opcode byte specify the fixed offset from A0, which means the fixed offset has a range of 0 to 7.

Opcodes that support indirect indexed addressing are LD, ST, ADD, XOR, AND, OR.

Examples include:

- LD (A0), 2
- ADD (A0), 3
- AND (A0), 4
- ST (A0), 7

9.2.4 Indirect indexed

In this form of addressing, an address register is used as a base address, and then an index register is used to offset from that base address to give the effective address.

The address register is one of 4, and is selected via bits 2-1 of the opcode byte as follows:

Table 12. Address register selection

Opcode bits 2-1	address register selected
00	A0
01	A1
10	A2
11	A3

Bit 0 of the opcode byte selects whether index register C1 or C2 is used:

The counter is selected as follows:

Table 13. Interpretation of counter for DBR

Opcode	Interpretion
0	C1
1	C2

Opcodes that support indirect indexed addressing are LD, ST, ADD, XOR.

Examples include:

- LD (A2), C1
- ADD (A1), C1
- ST (A3), C2

9.3 ADD - ADD TO ACCUMULATOR

Mnemonic: ADD

Opcode: `1010xxxx, and 11101xxx`

Usage: ADD effective-address, or ADD immediate-value

The ADD instruction adds the specified 32-bit value to the Accumulator via modulo 2^{32} addition.

The `11101xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 19). The `1010xxxx` form of the opcode defines an effective address as follows:

Table 14. Interpretation of operand for ADD (1010xxxx)

bit 3	Interpretion	Comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 20)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 20)

The Z flag is also set during this operation, depending on whether the result (loaded into the Accumulator) is zero or not.

9.4 AND - BITWISE AND

Mnemonic: AND

Opcode: `10000xxx, and 11010xxx`

Usage: AND effective-address, or AND immediate-value

The AND instruction performs a 32-bit bitwise AND operation on the Accumulator.

The `11010xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 19). The `10000xxx` form of the opcode follows the indirect fixed addressing rules (see Section 9.2.3 on page 20).

The Z flag is also set during this operation, depending on whether the resultant 32-bit value (loaded into the Accumulator) is zero or not.

9.5 DBR - DECREMENT AND BRANCH

Mnemonic: DBR

Opcode: 0011xxxx

Usage: DBR Counter, Offset

This instruction provides the mechanism for building simple loops.

The counter is selected from bit 0 of the opcode byte as follows:

Table 15. Interpretation of counter for DBR

Bit 0	Interpretation
0	C1
1	C2

If the specified counter is non-zero, then the counter is decremented and the designated offset is added to the current instruction address (PC for 1-byte instructions, PC+1 for 2-byte instructions). If the specified counter is zero, it is decremented (all bits in the counter become set) and processing continues at the next instruction (PC+1 or PC+2). The designated offset will typically be negative for use in loops.

The instruction is either 1 or two bytes, as determined by bits 3-1 of the opcode byte:

- If bits 3-1 = 000, the instruction consumes 2 bytes. The 8 bits at PC+1 are treated as a signed number and used as the offset amount. Thus 0xFF is treated as -1, and 0x01 is treated as +1.
- If bits 3-1 ≠ 000, the instruction consumes 1 byte. Bits 3-1 are treated as a negative number (the sign bit is implied) and used as the offset amount. Thus 111 is treated as -1, and 001 is treated as -7. This is useful for small loops.

The effect is that if the branch is back 1-7 bytes (1 byte is not particularly useful), then the single byte form of the instruction can be used. If the branch is forward, or backward more than 7 bytes, then the 2-byte instruction is required.

9.6 ERA - ERASE

Mnemonic: ERA

Opcode: 01110001

Usage: ERA

This instruction causes an erasure of the 256-byte page of flash memory pointed to by the Accumulator. The Accumulator is assumed to contain an 8-bit pointer to a 128-bit (16 byte) aligned structure (same structure as the address registers). The page number to be erased comes from bits 7-4, and the lower 4 bits are ignored.

Note that the size of the flash memory page being erased is actually 512 bytes, but in terms of data storage and addressing from the point of view of the CPU, there is only 256 bytes in the page.

9.7 HALT - HALT CPU OPERATION

Mnemonic: HALT

Opcode: 01110101

Usage: HALT

The HALT instruction writes a 0 to the internal GO register, thereby causing the CPU to terminate the currently executing program. The CPU will only be restarted with a new localId transaction from the Master or by a globalId plus Active Mode byte.

9.8 JMP - JUMP

Mnemonic: JMP

Opcode: 0000xxxx

Usage: JMP effective-address

The JMP instruction provides for a method of branching to a specified address. The instruction loads the PC with the effective address.

The new PC is loaded as follows: bits 11-8 are obtained from bits 3-0 of the JMP opcode byte, and bits 7-0 are obtained from PC+1.

9.9 JPI - JUMP INDIRECT

Mnemonic: JPI

Opcode: 01100011

Usage: JPI

The JPI instruction loads the PC with the lower 12 bits of the Accumulator, and sets the PCRamSel register with bit 15 of the Accumulator. Note that the stack is unaffected (unlike JSI).

9.10 JSI - JUMP SUBROUTINE INDIRECT

Mnemonic: JSI

Opcode: 01110011

Usage: JSI

The JSI instruction allows the jumping to a subroutine whose address is obtained from the Accumulator. The instruction pushes the current PC onto the stack, loads the PC with the lower 12 bits of the Accumulator, and sets the PCRamSel register with bit 15 of the Accumulator.

The stack provides for 12 levels of execution (11 subroutines deep). It is the responsibility of the programmer to ensure that this depth is not exceeded or the deepest return value will be overwritten (since the stack wraps). Programs can take advantage of the fact that the stack wraps.

9.11 JSR - JUMP SUBROUTINE

Mnemonic: JSR

Opcode: 0001xxxx

Usage: JSR effective-address

The JSR instruction provides for the most common usage of the subroutine construct. The instruction pushes the current PC onto the stack, and loads the PC with the effective address.

The new PC is loaded as follows: bits 11-8 are obtained from bits 3-0 of the JSR opcode byte, and bits 7-0 are obtained from PC+1.

The stack provides for 12 levels of execution (11 subroutines deep). It is the responsibility of the programmer to ensure that this depth is not exceeded or the return value will be overwritten (since the stack wraps). Programs can take advantage of the fact that the stack wraps.

9.12 LD - LOAD ACCUMULATOR

Mnemonic: LD

Opcode: `1011xxxx, and 11110xxx`

Usage: LD effective-address, or LD immediate-value

The LD instruction loads the Accumulator with the 32-bit value.

The `11110xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 19). The `1011xxxx` form of the opcode defines an effective address as follows:

Table 16. Interpretation of operand for LD (1011xxxx)

bit 3	Interpretation	Comment
0	(A0), offset	Indirect fixed addressing (see Section 9.2.3 on page 20)
1	(An), Cn	Indirect indexed addressing (see Section 9.2.4 on page 20)

The Z flag is also set during this operation, depending on whether the value loaded into the Accumulator is zero or not.

9.13 LIA - LOAD IMMEDIATE ADDRESS

Mnemonic: LIA

Opcode: `01111xxx`

Usage: LIAF AddressRegister, Value # for flash addresses
LIAR AddressRegister, Value # for ram addresses

The LIA instruction transfers the data from PC+1 into the designated address register (A0-A3), and sets the memory mode bit for that address register.

Bit 0 specifies whether the address is in flash or ram, as follows:

Table 17. Interpretation of memory mode for LIA

bit 0	Interpretation
0	flash
1	ram

The address register to be targetted is selected via bits 2-1 of the instruction.

9.14 OR - BITWISE OR

Mnemonic: OR

Opcode: `10001xxx, and 11011xxx`

Usage: OR effective-address, or OR immediate-value

The OR instruction performs a 32-bit bitwise OR operation on the Accumulator.

The `11011xxx` form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 19). The `10001xxx` form of the opcode follows the indirect fixed addressing rules (see Section 9.2.3 on page 20).

The Z flag is also set during this operation, depending on whether the resultant 32-bit value (loaded into the Accumulator) is zero or not.

9.15 RIA - ROTATE IN ADDRESS

Mnemonic: RIA

Opcode: `11111xxx`

Usage: RIAF AddressRegister # for flash addresses
RIAR AddressRegister # for ram addresses

The RIA instruction transfers the lower 8 bits of the Accumulator into the designated address register (A0-A3), sets the memory mode bit for that address register, and rotates the Accumulator right by 8 bits.

Bit 0 specifies whether the address is in flash or ram, as follows:

Table 18. Interpretation of memory mode for RIA

bit 0	interpretation
0	flash
1	ram

The address register to be targetted is selected via bits 2-1 of the instruction.

9.16 ROR - ROTATE RIGHT

Mnemonic: ROR

Opcode: 1100xxxx

Usage: ROR Value

The ROR instruction provides a way of rotating the Accumulator right a set number of bits. The bit(s) coming in at the top of the Accumulator (to become bit 31) can either come from the previous lower bits of the Accumulator, from the serial connection, or from external flags. The bit(s) rotated out can also be output from the serial connection, or combined with an external flag.

The allowed operands are as follows:

Table 19. Interpretation of operand for ROR

Bits 3-0	Interpretation
0000	RB
0001	XRB
0010	WriteMask
0011	1
0100	- (reserved)
0101	3
0110	31
0111	24
1000	C1
1001	C2
1010	- (reserved)
1011	- (reserved)
1100	8
1101	ID
1110	InByte
1111	OutByte

The Z flag is also set during this operation, depending on whether resultant 32-bit value (loaded into the Accumulator) is zero or not.

In its simplest form, the operand for the ROR instruction is one of 1, 3, 8, 24, 31, indicating how many bit positions the Accumulator should be rotated. For these operands, there is no external input or output - the bits of the Accumulator are merely rotated right. Note that these values are the equivalent to rotating left 31, 29, 24, 8, 1 bit positions.

With operand WriteMask, the lower 8 bits of the Accumulator are transferred to the WriteMask register, and the Accumulator is rotated right by 1 bit. This conveniently allows successive nybbles to be masked during Flash writes if the Accumulator has been preloaded with an appropriate value (eg 0x01).

With operands C1 and C2, the lower appropriate number of bits of the Accumulator (3 for C1, 6 for C2) are transferred to the C1 or C2 register and the lower 6 bits of the Accumulator are loaded with the previous value of the Cn register. The remaining upper bits of the Accumulator are set as follows: bit 31-24 are copied from previous bits 7-0, and bits 23-6 are

copied from previous bits 31-14 (effectively junk). As a result, the Accumulator should be subsequently masked if the programmer wants to compare for specific values).

With operand ID, the 6 low-order bits are transferred from the Accumulator to the LocalId register, the low-order 8 bits of the Accumulator are copied to the Trim register if the Trim register has not already been written to after power-on reset, and the Accumulator is rotated right by 8 bits. This means that the ROR ID instruction needs to be performed twice, typically during Global Active Mode - once to set Trim, and once to set LocalId. *Note there is no way to read the contents of the localId or Trim registers directly.*

With operand InByte, the next serial input byte is transferred to the highest 8 bits of the Accumulator. The InByteValid bit is also cleared. If there is no input bit available from the client yet, execution is suspended until there is one. The remainder of the Accumulator is shifted right 8 bit positions (bit31 becomes bit 23 etc.), with lowest bits of the Accumulator shifted out.

With operand OutByte, the Accumulator is shifted right 8 bit positions. The byte shifted out from bits 7-0 is stored in the OutByte register and the OutByteValid flag is set. It is therefore ready for a client to read. If the OutByteValid flag is already set, execution of the instruction stalls until the OutByteValid flag cleared (when the OutByte byte has been read by the client). The new data shifted in to the upper 8 bits of the Accumulator is what was transferred to the OutByte register (i.e. from the Accumulator).

Finally, the RB and XRB operands allow the implementation of LFSRs and multiple precision shift registers. With RB, the bit shifted out (formally bit 0) is written to the RTMP register. The register currently in the RTMP register becomes the new bit 31 of the Accumulator. Performing multiple ROR RB commands over several 32-bit values implements a multiple precision rotate/shift right. The XRB operates in the same way as RB, in that the current value in the RTMP register becomes the new bit 31 of the Accumulator. However with the XRB instruction, the bit formally known as bit 0 does not simply replace RTMP (as in the RB instruction). Instead, it is XORed with RTMP, and the result stored in RTMP. This allows the implementation of long LFSRs, as required by the authentication protocol.

9.17 RST - RESET

Mnemonic: RST
Opcode: 01100010
Usage: RST

The RST instruction loads the PC with 0.

Programmers will not typically use the RST command. However the CPU executes this instruction whenever a new command arrives over the serial interface, so that the code entry point is known i.e. every time the chip receives a new command, execution begins at address 0.

9.18 RTS - RETURN FROM SUBROUTINE

Mnemonic: RTS
Opcode: 01110100
Usage: RTS

The RTS instruction pulls the saved PC from the stack, adds 1, and resumes execution at the resultant address. The effect is to cause execution to resume at the instruction after the most recently executed JSR or JSI instruction.

Although 12 levels of execution are provided for (11 subroutines), it is the responsibility of the programmer to balance each JSR and JSI instruction with an RTS. A RTS executed with no previous JSR will cause execution to begin at whatever address happens to be pulled from the stack. Of course this may be desired behaviour in specific circumstances.

9.19 SC - SET COUNTER

Mnemonic: SC

Opcode: 0100xxxx

Usage: SC Counter Value

The SC instruction is used to transfer a 3-bit Value into the specified counter. The operand determines which of counters C1 and C2 is to be loaded as well as the value to be loaded. Value is stored in bits 3-1 of the 8-bit opcode, and the counter is specified by bit 0 as follows:

Table 20. Interpretation of counter for SC

bit0	Interpretion
0	C1
1	C2

Since counter C1 is 3 bits, Value is copied directly into C1.

For counter C2, $C2_{2-0}$ are copied to $C2_{5-3}$, and Value is copied to $C2_{2-0}$. Two SC C2 instructions are therefore required to load C2 with a given 6-bit value. For example, to load C2 with 0x0C, we would have SC C2 1 followed by SC C2 4.

9.20 ST - STORE ACCUMULATOR

Mnemonic: ST

Opcode: 0101xxxx

Usage: ST effective-address

The ST instruction stores the 32-bit Accumulator at the effective address. The effective address is determined as follows:

Table 21. Interpretation of operand for ST (0101xxxx)

bit3	Interpretion	comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 20)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 20)

If the effective address in Flash memory, only those nybbles whose corresponding Write-Mask bit is set will be written to Flash. Programmers should be very aware of flash characteristics (write time, longevity, page size etc. when storing data in flash).

There is always the possibility that power could be removed during a write to Flash. If this occurs, the flash will be in an indeterminate state. The QA Chip can be warned by the sys-

tem that power is about to be removed (via the master sending 3 stop bits in a row), and therefore will not attempt to perform the write. Therefore from a programming point of view, a write can be considered to be atomic (i.e. always succeeds).

9.21 TBR - TEST AND BRANCH

Mnemonic: TBR

Opcode: 0010xxxx

Usage: TBR Value Offset

The Test and Branch instruction tests the status of the Z flag (the zero-ness of the Accumulator), and then branches if a match ocurs.

The zero-ness is selected from bit 0 of the opcode byte as follows:

Table 22. Interpretation of zero-ness for TBR

bit 0	interpretation
0	true if Acc is zero (Z = 1)
1	true if Acc is non-zero (Z=0)

If the specified zero-test matches, then the designated offset is added to the current instruction address (PC for 1-byte instructions, PC+1 for 2-byte instructions). If the zero-test does not match, processing continues at the next instruction (PC+1 or PC+2). The instruction is either 1 or two bytes, as determined by bits 3-1 of the opcode byte:

- If bits 3-1 = 000, the instruction consumes 2 bytes. The 8 bits at PC+1 are treated as a signed number and used as the offset amount to be added to PC+1. Thus 0xFF is treated as -1, and 0x01 is treated as +1.
- If bits 3-1 ≠ 000, the instruction consumes 1 byte. Bits 3-1 are treated as a positive number (the sign bit is implied) and used as the offset amount to be added to PC. Thus 111 is treated as 7, and 001 is treated as 1. This is useful for skipping over a small number of instructions.

The effect is that if the branch is forward 1-7 bytes (1 byte is not particularly useful), then the single byte form of the instruction can be used. If the branch is backward, or forward more than 7 bytes, then the 2-byte instruction is required.

9.22 XOR - BITWISE EXCLUSIVE OR

Mnemonic: XOR

Opcode: 1001xxxx, and 11100xxx

Usage: XOR effective-address, or XOR immediate-value

The XOR instruction performs a 32-bit bitwise XOR operation on the Accumulator.

The 11100xxx form of the opcode follows the immediate addressing rules (see Section 9.2.1 on page 19). The 1001xxxx form of the opcode has an effective address as follows:

Table 23. Interpretation of operand for XOR (1001xxxx)

bit 0	interpretation	comment
0	(A0), offset	indirect fixed addressing (see Section 9.2.3 on page 20)
1	(An), Cn	indirect indexed addressing (see Section 9.2.4 on page 20)

The Z flag is also set during this operation, depending on whether the result (loaded into the Accumulator) is zero or not.

IMPLEMENTATION

10 Introduction

This chapter provides the high-level definition of a purpose-built CPU capable of implementing the functionality required of an QA Chip.

Note that this CPU is not a general purpose CPU. It is tailor-made for implementing the authentication logic. The authentication commands that a user of an QA Chip sees, such as TST, RND etc. are all implemented as small programs written in the CPU instruction set.

10.1 PHYSICAL INTERFACE

10.1.1 Pin connections

The pin connections are described in Table 24.

Table 24. Pin connections to QA Chip

pin	direction	description
Vdd	In	Nominal voltage. If the voltage deviates from this by more than a fixed amount, the chip will RESET.
GND	In	
SClk	In	Serial clock
SDa	In/Out	Serial data

The system operating clock SysClk is different to SClk. SysClk is derived from an internal ring oscillator based on the process technology. In the FPGA implementation SysClk is obtained via a 5th pin.

10.1.2 Size and cost

The QA Chip uses a 0.25 µm CMOS Flash process for an area of 1mm^2 yielding a 10 cent manufacturing cost in 2002. A breakdown of area is listed in Table 25.

Table 25. Breakdown of Area for QA Chip

approximate area (mm^2)	description
0.49	8KByte flash memory TSMC: SFC0008_08B9_HE (8K x 8-bits, erase page size = 512 bytes) Area = 724.688µm x 682.05 µm.
0.08	3072 bits of static RAM
0.38	General logic
0.05	Analog circuitry
	TOTAL (approximate)

Note that there is no specific test circuitry (scan chains or BIST) within the QA Chip (see Section 10.3.10 on page 39), so the total transistor count is as shown in Table 25.

10.1.3 Reset

The chip performs a RESET upon power-up. In addition, tamper detection and prevention circuitry in the chip will cause the chip to either RESET or erase Flash memory (depending on the attack detected) if an attack is detected.

10.2 OPERATING SPEED

The base operating system clock SysClk is generated internally from a ring oscillator (process dependant). Since the frequency varies with operating temperature and voltage, the clock is passed through a temperature-based clock filter before use (see Section 10.3.3 on page 34). The frequency is built into the chip during manufacture, and cannot be changed. The frequency is recommended to be about 4-10 MHz.

10.3 GENERAL MANUFACTURING COMMENTS

Manufacturing comments are not normally made when normally describing the architecture of a chip. However, in the case of the QA Chip, the physical implementation of the chip is very much tied to the security of the key. Consequently a number of specialized circuits and components are necessary for implementation of the QA Chip. They are listed here.

- Flash process
- Internal randomized clock
- Temperature based clock filter
- Noise generator
- Tamper Prevention and Detection circuitry
- Protected memory with tamper detection
- Boot-strap circuitry for loading program code
- Data connections in polysilicon layers where possible
- OverUnderPower Detection Unit
- No scan-chains or BIST

10.3.1 Flash process

The QA Chip is implemented with a standard Flash manufacturing process. It is important that a Flash process be used to ensure that good endurance is achieved (parts of the Flash memory can be erased/written many times).

10.3.2 Internal randomized clock

To prevent clock glitching and external clock-based attacks, the operating clock of the chip should be generated internally. This can be conveniently accomplished by an internal ring oscillator. The length of the ring depends on the process used for manufacturing the chip.

Due to process and temperature variations, the clock needs to be trimmed to bring it into a range usable for timing of Flash memory writes and erases.

The internal clock should also contain a small amount of randomization to prevent attacks where light emissions from switching events are captured, as described below.

Finally, the generated clock must be passed through a temperature-based clock filter before being used by the rest of the chip (see Section 10.3.3 on page 34).

The normal situation for FET implementation for the case of a CMOS inverter (which involves a pMOS transistor combined with an nMOS transistor) as shown in Figure 1:

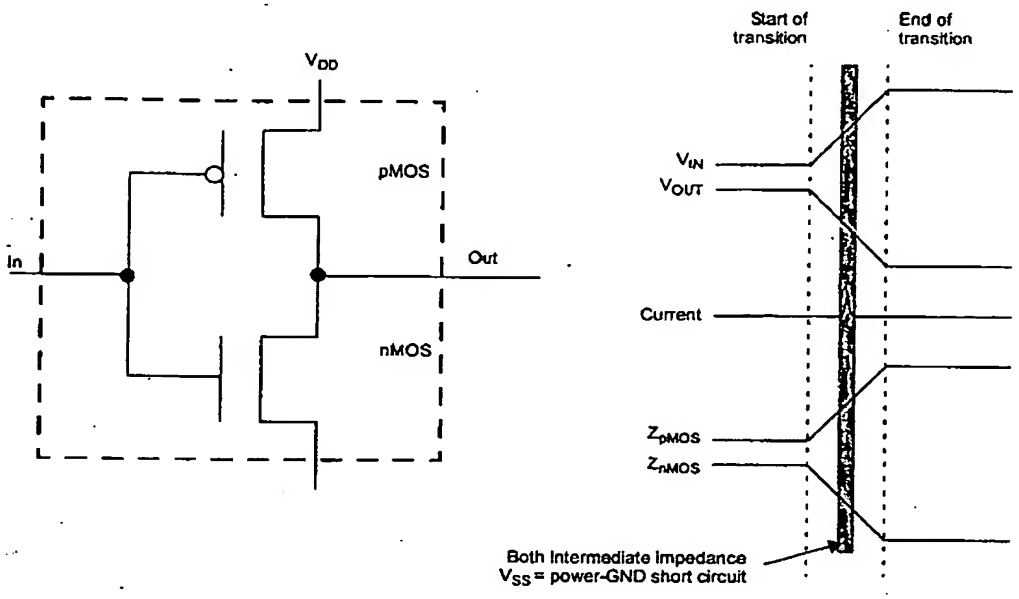


Figure 1. Normal FET Implementation of CMOS Inverter

During the transition, there is a small period of time where both the nMOS transistor and the pMOS transistor have an intermediate resistance. The resultant power-ground short circuit causes a temporary increase in the current, and in fact accounts for around 20% of current consumed by a CMOS device. A small amount of infrared light is emitted during the short circuit, and can be viewed through the silicon substrate (silicon is transparent to infrared light). A small amount of light is also emitted during the charging and discharging of the transistor gate capacitance and transmission line capacitance.

For circuitry that manipulates secret key information, such information must be kept hidden.

Fortunately, IBM's PICA system and LVP (laser voltage probe) both have a requirement for repeatability due to the fact that the photo emissions are extremely weak (one photon requires more than 10^5 switching events). PICA requires around 10^9 pases to build a picture of the optical waveform. Similarly the LVP requires multiple passes to ensure an adequate SNR.

Randomizing the clock stops repeatability (from the point of view of collecting information about the same position in time), and therefore reduces the possibility of this attack.

10.3.3 Temperature based clock filter

The QA Chip circuitry is designed to operate within a specific clock speed range. Although the clock is generated by an internal ring oscillator, the speed varies with temperature and power. Since the user supplies the temperature and power, it is possible for an attacker to attempt to introduce race-conditions in the circuitry at specific times during processing. An example of this is where a low temperature causes a clock speed higher than the circuitry is designed for, and this may prevent an XOR from working properly,

and of the two inputs, the first may always be returned. These styles of transient fault attacks are documented further in [1]. The lesson to be learned from this is that the input power and operating temperature *cannot be trusted.*

Since the chip contains a specific power filter, we must also filter the clock. This can be achieved with a temperature sensor that allows the clock pulses through only when the temperature range is such that the chip can function correctly.

The filtered clock signal would be further divided internally as required.

10.3.4 Noise Generator

Each QA Chip should contain a noise generator that generates continuous circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and add noise to the I_{dd} signal. Placement of the noise generator is not an issue on an QA Chip due to the length of the emission wavelengths.

The noise generator is used to generate electronic noise, multiple state changes each clock cycle, and as a source of pseudo-random bits for the Tamper Prevention and Detection circuitry (see Section 10.3.5 on page 35).

A simple implementation of a noise generator is a 64-bit maximal period LFSR seeded with a non-zero number.

10.3.5 Tamper Prevention and Detection circuitry

A set of circuits is required to test for and prevent physical attacks on the QA Chip. However what is actually detected as an attack may not be an intentional physical attack. It is therefore important to distinguish between these two types of attacks in an QA Chip:

- where you *can be certain* that a physical attack has occurred.
- where you *cannot* be certain that a physical attack has occurred.

The two types of detection differ in what is performed as a result of the detection. In the first case, where the circuitry can be certain that a true physical attack has occurred, erasure of flash memory key information is a sensible action. In the second case, where the circuitry cannot be sure if an attack has occurred, there is still certainly something wrong. Action must be taken, but the action should not be the erasure of secret key information. A suitable action to take in the second case is a chip RESET. If what was detected was an attack that has permanently damaged the chip, the same conditions will occur next time and the chip will RESET again. If, on the other hand, what was detected was part of the normal operating environment of the chip, a RESET will not harm the key.

A good example of an event that circuitry cannot have knowledge about, is a power glitch. The glitch may be an intentional attack, attempting to reveal information about the key. It may, however, be the result of a faulty connection, or simply the start of a power-down sequence. It is therefore best to only RESET the chip, and not erase the key. If the chip was powering down, nothing is lost. If the System is faulty, repeated RESETs will cause the consumer to get the System repaired. In both cases the consumable is still intact.

A good example of an event that circuitry can have knowledge about, is the cutting of a data line within the chip. If this attack is somehow detected, it could only be a result of a faulty chip (manufacturing defect) or an attack. In either case, the erasure of the secret information is a sensible step to take.

Consequently each QA Chip should have 2 Tamper Detection Lines - one for definite attacks, and one for possible attacks. Connected to these Tamper Detection Lines would be a number of Tamper Detection test units, each testing for different forms of tampering. *In addition, we want to ensure that the Tamper Detection Lines and Circuits themselves cannot also be tampered with.*

At one end of the Tamper Detection Line is a source of pseudo-random bits (clocking at high speed compared to the general operating circuitry). The Noise Generator circuit described above is an adequate source. The generated bits pass through two different paths - one carries the original data, and the other carries the inverse of the data. The wires carrying these bits are in the layer above the general chip circuitry (for example, the memory, the key manipulation circuitry etc.). The wires must also cover the random bit generator. The bits are recombined at a number of places via an XOR gate. If the bits are different (they should be), a 1 is output, and used by the particular unit (for example, each output bit from a memory read should be ANDed with this bit value). The lines finally come together at the Flash memory Erase circuit, where a complete erasure is triggered by a 0 from the XOR. Attached to the line is a number of triggers, each detecting a physical attack on the chip. Each trigger has an oversize nMOS transistor attached to GND. The Tamper Detection Line physically goes through this nMOS transistor. If the test fails, the trigger causes the Tamper Detect Line to become 0. The XOR test will therefore fail on either this clock cycle or the next one (on average), thus RESETing or erasing the chip.

Figure 2 illustrates the basic principle of a Tamper Detection Line in terms of tests and the XOR connected to either the Erase or RESET circuitry.

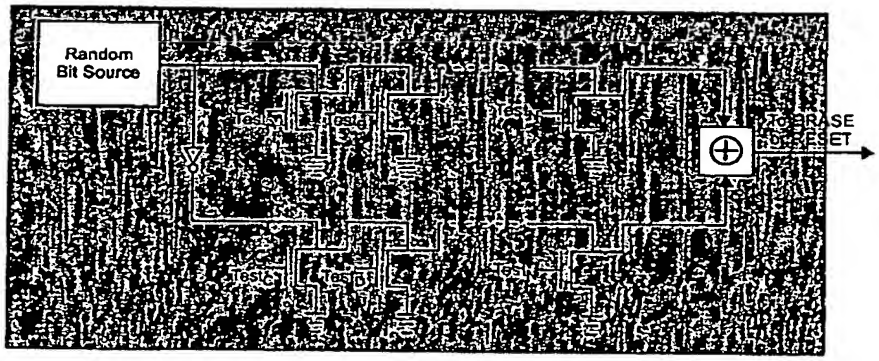


Figure 2. Tamper Detection Line

The Tamper Detection Line must go through the drain of an output transistor for each test, as illustrated by Figure 3:

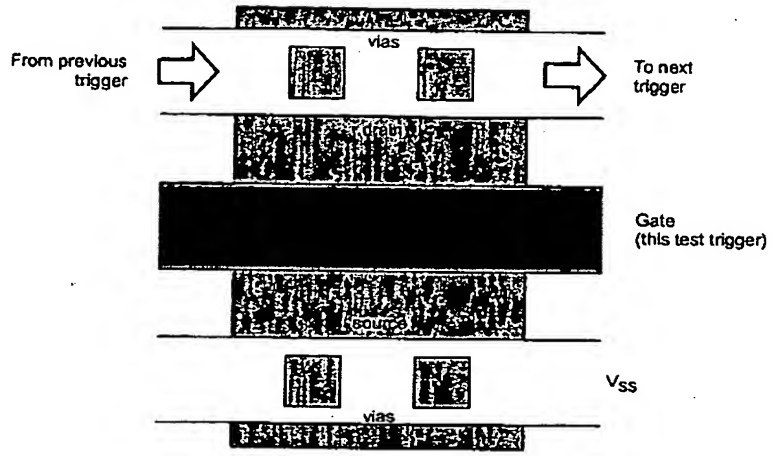


Figure 3. Oversize nMOS Transistor Layout of Tamper Detection Line

It is not possible to break the Tamper Detect Line since this would stop the flow of 1s and 0s from the random source. The XOR tests would therefore fail. As the Tamper Detect Line physically passes through each test, it is not possible to eliminate any particular test without breaking the Tamper Detect Line.

It is important that the XORs take values from a variety of places along the Tamper Detect Lines in order to reduce the chances of an attack. Figure 4 illustrates the taking of multiple XORs from the Tamper Detect Line to be used in the different parts of the chip. Each of these XORs can be considered to be generating a ChipOK bit that can be used within each unit or sub-unit.

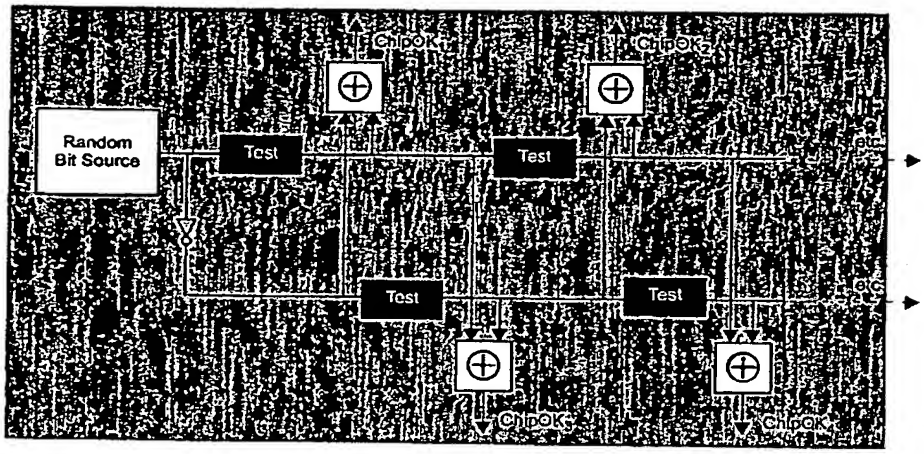


Figure 4. Multiple XORs along the Tamper Detection Line

A typical usage would be to have an OK bit in each unit that is ANDed with a given ChipOK bit each cycle. The OK bit is loaded with 1 on a RESET. If OK is 0, that unit will fail until the next RESET. If the Tamper Detect Line is functioning correctly, the chip will

either RESET or erase all key information. If the RESET or erase circuitry has been destroyed, then this unit will not function, thus thwarting an attacker.

The destination of the RESET and Erase line and associated circuitry is very context sensitive. It needs to be protected in much the same way as the individual tamper tests. There is no point generating a RESET pulse if the attacker can simply cut the wire leading to the RESET circuitry. The actual implementation will depend very much on what is to be cleared at RESET, and how those items are cleared.

Finally, Figure 5 shows how the Tamper Lines cover the noise generator circuitry of the chip. The generator and NOT gate are on one level, while the Tamper Detect Lines run on a level above the generator.

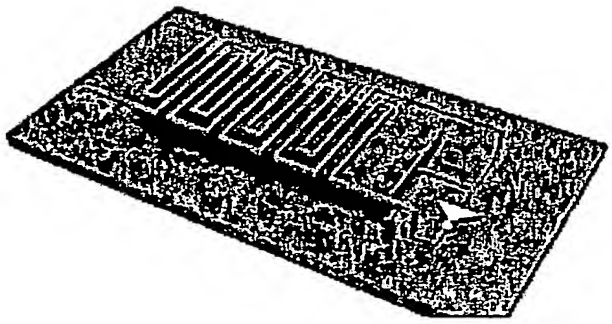


Figure 5. Tamper Detection Lines Cover the Noise Generator

10.3.6 Protected memory with tamper detection

It is not enough to simply store secret information or program code in flash memory. The Flash memory and RAM must be protected from an attacker who would attempt to modify (or set) a particular bit of program code or key information. The mechanism used must conform to being used in the Tamper Detection Circuitry (described above).

The first part of the solution is to ensure that the Tamper Detection Line passes directly above each flash or RAM bit. This ensures that an attacker cannot probe the contents of flash or RAM. A breach of the covering wire is a break in the Tamper Detection Line. The breach causes the Erase signal to be set, thus deleting any contents of the memory. The high frequency noise on the Tamper Detection Line also obscures passive observation.

The second part of the solution for flash is to always store the data with its inverse. In each byte, 4 bits contains the data, and 4 bits (the shadow) contains the inverse of the data. If both are 0, this is a valid erase state, and the value is 0. Otherwise, the memory is only valid if the 4 bits of shadow are the inverse of the main 4 bits. The reasoning is that it is possible to add electrons to flash via a FIB, but not take electrons away. If it is possible to change a 0 to 1 for example, it is not possible to do the same to its inverse, and therefore regardless of the sense of flash, an attack can be detected.

The second part of the solution for RAM is to use a parity bit. The data part of the register can be checked against the parity bit (which will not match after an attack).

The bits coming from Flash and RAM can therefore be validated by a number of test units (one per bit) connected to the common Tamper Detection Line. The Tamper Detection cir-

cuitry would be the first circuitry the data passes through (thus stopping an attacker from cutting the data lines).

In addition, the data and program code should be stored in different locations for each chip, so an attacker does not know where to launch an attack. Finally, XORing the data coming in and going to Flash with a random number that varies for each chip means that the attacker cannot learn anything about the key by setting or clearing an individual bit that has a probability of being the key (the inverse of the key must also be stored somewhere in flash).

Finally, each time the chip is called, every flash location is read before performing any program code. This allows the flash tamper detection to be activated in a common spot instead of when the data is actually used or program code executed. This reduces the ability of an attacker to know exactly what was written to.

10.3.7 Boot-strap circuitry for loading program code

Program code should be kept in protected flash instead of ROM, since ROM is subject to being altered in a non-testable way. A boot-strap mechanism is therefore required to load the program code into flash memory (flash memory is in an indeterminate state after manufacture).

The boot-strap circuitry must not be in a ROM - a small state-machine suffices. Otherwise the boot code could be trivially modified in an undetectable way.

The boot-strap circuitry must erase all flash memory, check to ensure the erasure worked, and then load the program code.

The program code should only be executed once the flash program memory has been validated via Program Mode.

Once the final program has been loaded, a fuse can be blown to prevent further programming of the chip.

10.3.8 Connections in polysilicon layers where possible

Wherever possible, the connections along which the key or secret data flows, should be made in the polysilicon layers. Where necessary, they can be in metal 1, but must never be in the top metal layer (containing the Tamper Detection Lines).

10.3.9 OverUnder Power Detection Unit

Each QA Chip requires an OverUnder Power Detection Unit (PDU) to prevent Power Supply Attacks. A PDU detects power glitches and tests the power level against a Voltage Reference to ensure it is within a certain tolerance (TBD). The Unit contains a single Voltage Reference and two comparators. The PDU would be connected into the RESET Tamper Detection Line, thus causing a RESET when triggered.

A side effect of the PDU is that as the voltage drops during a power-down, a RESET is triggered, thus erasing any work registers.

10.3.10 No scan chains or BIST

Test hardware on an QA Chip could very easily introduce vulnerabilities. In addition, due to the small size of the QA Chip logic, test hardware such as scan paths and BIST units

could in fact take a sizeable chunk of the final chip, lowering yield and causing a situation where an error in the test hardware causes the chip to be unusable. As a result, the QA Chip should not contain any BIST or scan paths. Instead, the program memory must first be validated via the Program Mode mechanism, and then a series of program tests run to verify the remaining parts of the chip.

11 Architecture

Figure 6 shows a high level block diagram of the QA Chip. Note that the tamper prevention and detection circuitry is not shown.

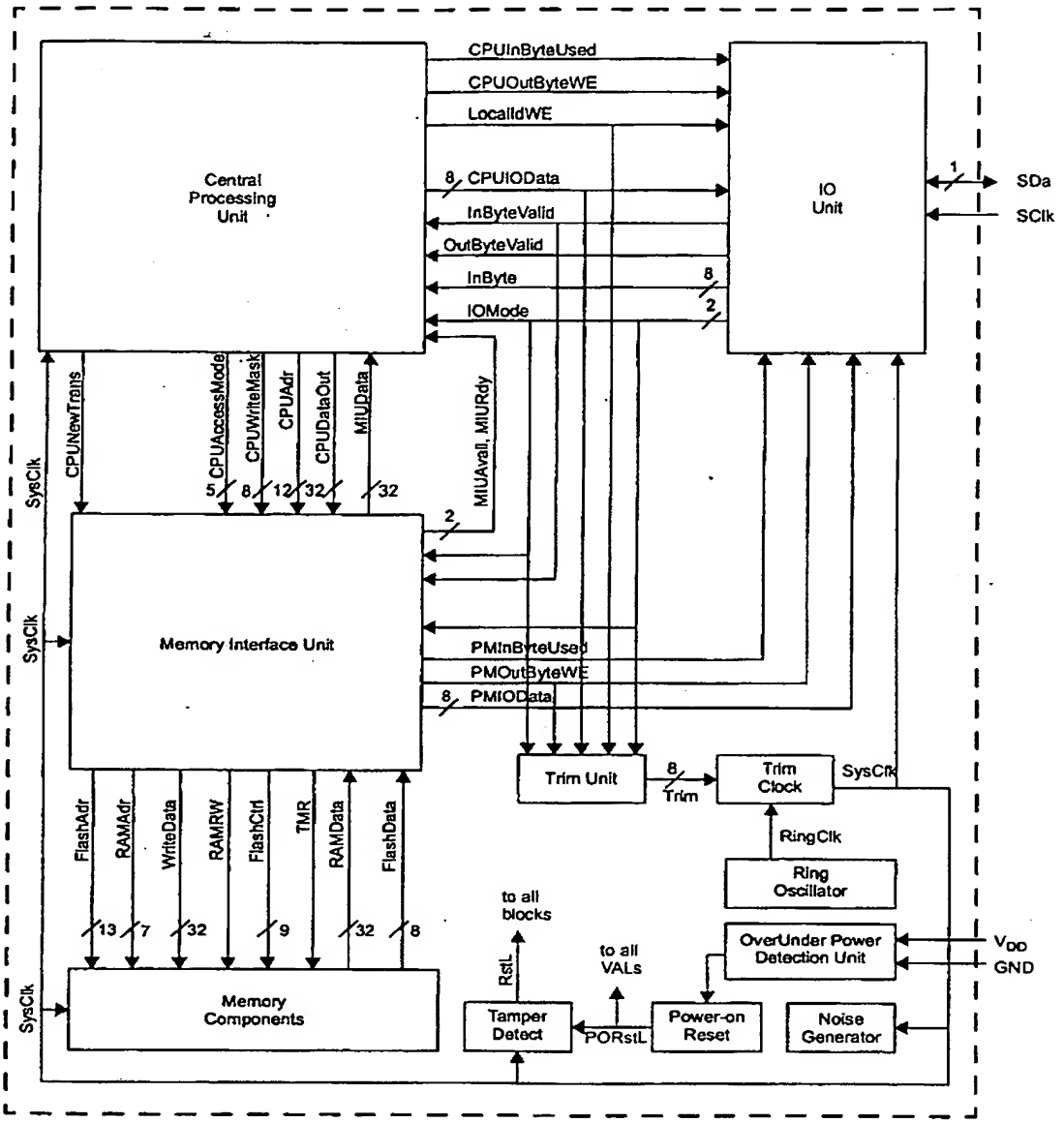


Figure 6. High level block diagram of QA Chip

11.1 RING OSCILLATOR

The operating clock of the chip (SysClk) is generated by an internal ring oscillator and passed through the Trim Unit before being used by the rest of the chip. The length of the ring depends on the process used for manufacturing the chip. A nominal operating frequency range of 10 MHz should be sufficient. This clock should contain a small amount of randomization to prevent attacks where light emissions from switching events are captured.

Note that this is different to the input SClk which is the serial clock for external communication.

The SysClk signal is not described any further within this document.

The ring oscillator is covered by both Tamper Detection and Prevention lines so that if an attacker attempts to tamper with the unit, the chip will either RESET or erase all secret information.

FPGA Note: the FPGA does not have an internal ring oscillator. An additional pin (SysClk) is used instead. This will be replaced by an internal ring oscillator in the final ASIC.

11.2 TRIM UNIT AND TRIM CLOCK

The Trim Clock block is an analog circuit that trims the ring oscillator output to bring it from 4:1 variation (due to process and temperature) down to 2:1 (temperature variations only) in order to satisfy the timing requirements of the Flash memory. The 8-bit Trim register within the Trim Unit has a reset value of 0x80, and is written to either by the PMU during Trim Mode or by the CPU in Active Mode. Note that the CPU is only able to write *once* to the Trim register between power-on-reset due to the TrimDone flag which provides overloading of LocalIdWE.

The frequency of the ring oscillator is measured by counting cycles[1], in the PMU, over the byte period of the serial interface. The frequency of the serial clock, SClk, and therefore the byte period will be accurately controlled during the measurement. The cycle count (Fmeas) at the end of the period is read over the serial bus and the Trim register updated (Trimval) from its power on default (POD) value. The steps are shown in Figure 7. Multi-

1. Note that the PMU counts using 12-bits, saturates at 0xFFF, and returns the cycle count divided by 2 as an 8-bit value. This means that multiple measure-read-trim cycles may be necessary to resolve any ambiguity. In any case, multiple cycles are necessary to test the correctness of the trim circuitry during manufacture test.

ple measure - read - trim cycles are possible to improve the accuracy of the trim procedure.

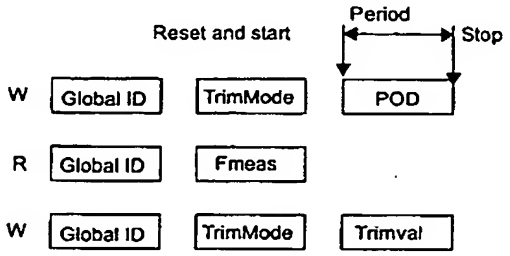


Figure 7. Serial bus protocol for trimming

A single byte for both Fmeas and Trimval provide sufficient accuracy for measurement and trimming of the frequency. If the bus operates at 400kHz, a byte (8 bits) can be sent in 20µs. By dividing the maximum oscillator frequency, expected to be 20MHz, by 2 results in a cycle count of 200 and 50 for the minimum frequency of 5MHz resulting in a worst case accuracy of 2%.

Figure 8 shows a block diagram of the Trim Unit:

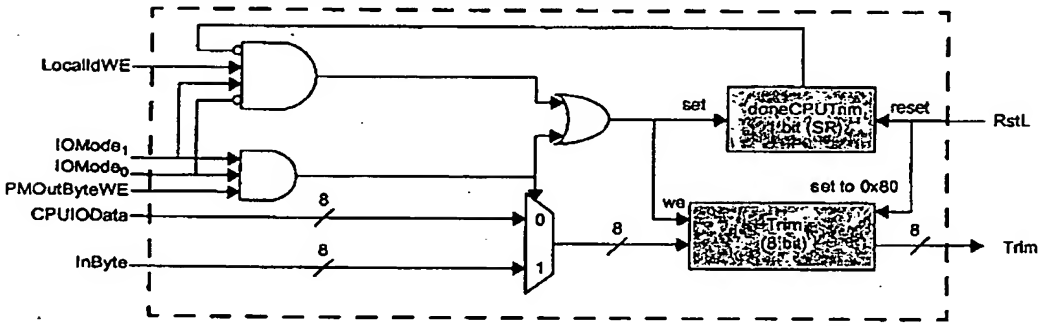


Figure 8. Block diagram of Trim Unit

The 8-bit Trim value is used in the analog Trim Block to adjust the frequency of the ring oscillator by controlling its bias current. Assuming a linear control, 8 bits will give a trim resolution of about 60kHz or 1.2%.

The analog Trim Clock circuit also contains a Temperature filter as described in Section 10.3.3 on page 34.

11.3 OVERUNDER POWER DETECTION UNIT

The OverUnder Power Detection Unit (PDU) is the same as that described in Section 10.3.9 on page 39.

Note that the PDU triggers the RESET Tamper Detection Line only. It does not trigger the Erase Tamper Detection Line.

The PDU can be implemented with regular CMOS, since the key does not pass through this unit. It does not have to be implemented with non-flashing CMOS.

The PDU is covered by both Tamper Detection and Prevention lines so that if an attacker attempts to tamper with the unit, the chip will either RESET or erase all secret information.

11.4 POWER-ON RESET AND TAMPER DETECT UNIT

The Power-on Reset unit (POR) detects a power-on condition and generates the PORstL signal that is fed to all the validation units, including the two inside the Tamper Detect Unit (TDU).

All other logic is connected to RstL, which is the PORstL gated by the VAL unit attached to the Reset tamper detection lines (see Section 10.3.5 on page 35) within the TDU. Therefore, if the Reset tamper line is asserted, the validation will drive RstL low, *and can only be cleared by a power-down*. If the tamper line is not asserted, then RstL = PORstL.

The TDU contains a second VAL unit attached to the Erase tamper detection lines (see Section 10.3.5 on page 35) within the TDU. It produces a TamperEraseOK signal that is output to the MIU (1 = the tamper lines are all OK, 0 = force an erasure of Flash).

11.5 NOISE GENERATOR

The Noise Generator (NG) is the same as that described in Section 10.3.4 on page 35. It is based on a 64-bit maximal period LFSR loaded with a set non-zero bit pattern on RESET.

The NG must be protected by both Tamper Detection and Prevention lines so that if an attacker attempts to tamper with the unit, the chip will either RESET or erase all secret information.

In addition, the bits in the LFSR must be validated to ensure they have not been tampered with (i.e. a parity check). If the parity check fails, the Erase Tamper Detection Line is triggered.

Finally, all 64 bits of the NG are ORed into a single bit. If this bit is 0, the Erase Tamper Detection Line is triggered. This is because 0 is an invalid state for an LFSR.

11.6 IO UNIT

The QA Chip acts as a *slave* device, accepting serial data from an external master via the IO Unit (IOU). Although the IOU actually transmits data over a 1-bit line, the data is always transmitted and received in 1-byte chunks.

The IOU receives commands from the master to place it in a specific operating mode, which is one of:

- **Idle Mode**: is the startup mode for the IOU. *Idle Mode* is the mode where the QA Chip is waiting for the next command from the master. Input signals from the CPU are ignored.
- **Program Mode**: is where the QA Chip erases all currently stored data in the Flash memory (program and secret key information) and then allows new data to be written to the Flash. The IOU stays in *Program Mode* until told to enter another mode.
- **Active Mode**: is where the QA Chip allows the program code to be executed to process the master's specific command. The IOU returns to *Idle Mode* automatically when the command has been processed, or if the time taken between consuming input

bytes (while the master is writing the data) or generating output bytes (while the master is reading the results) is too great.

- **Trim Mode**: is where the QA Chip allows the generation and setting of a trim value to be used on the internal ring oscillator clock value. This is required before any program can be stored in the Flash memory.

See Section 12 on page 49 for detailed information about the IOU.

11.7 CENTRAL PROCESSING UNIT

The Central Processing Unit (CPU) block provides the majority of the circuitry of the 4-bit microprocessor. Figure 9 shows a high level view of the block.

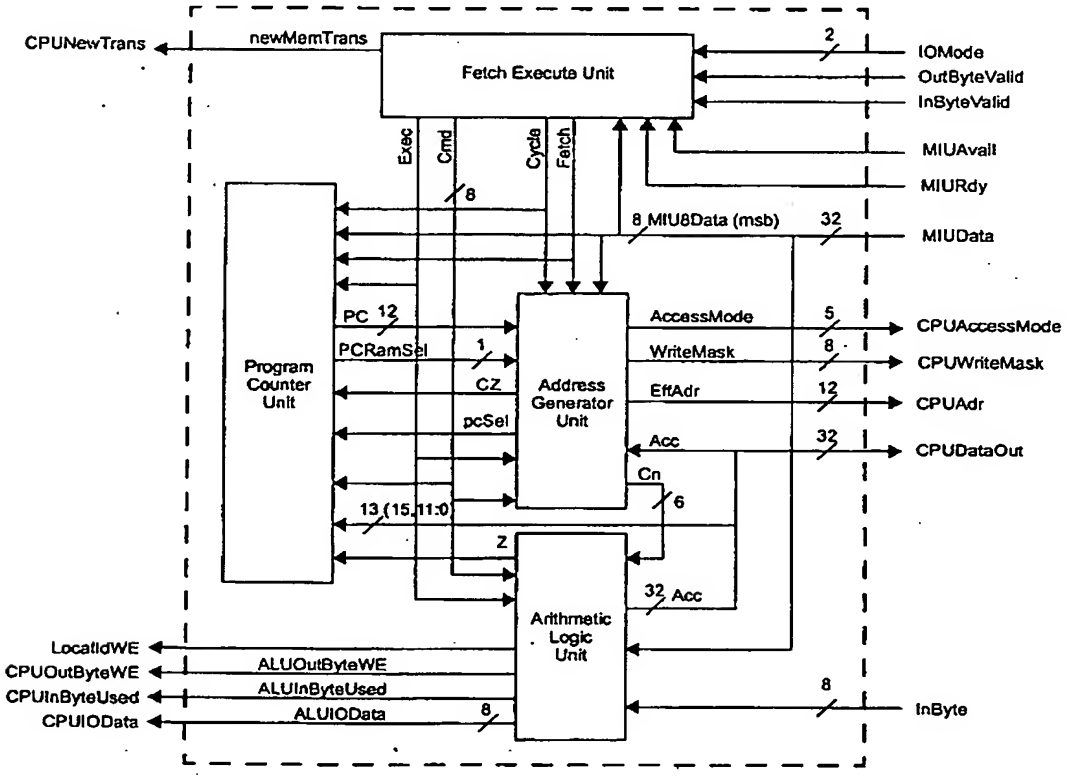


Figure 9. Block diagram of CPU

11.8 MEMORY INTERFACE UNIT

The Memory Interface Unit (MIU) provides the interface to flash and RAM. The MIU contains a Program Mode Unit that allows flash memory to be loaded via the IOU, a Memory Request Unit that maps 8-bit and 32-bit requests into multiple byte based requests, and a Memory Access Unit that generates read/write strobes for individual accesses to the memory.

Figure 10 shows a high level view of the MIU block.

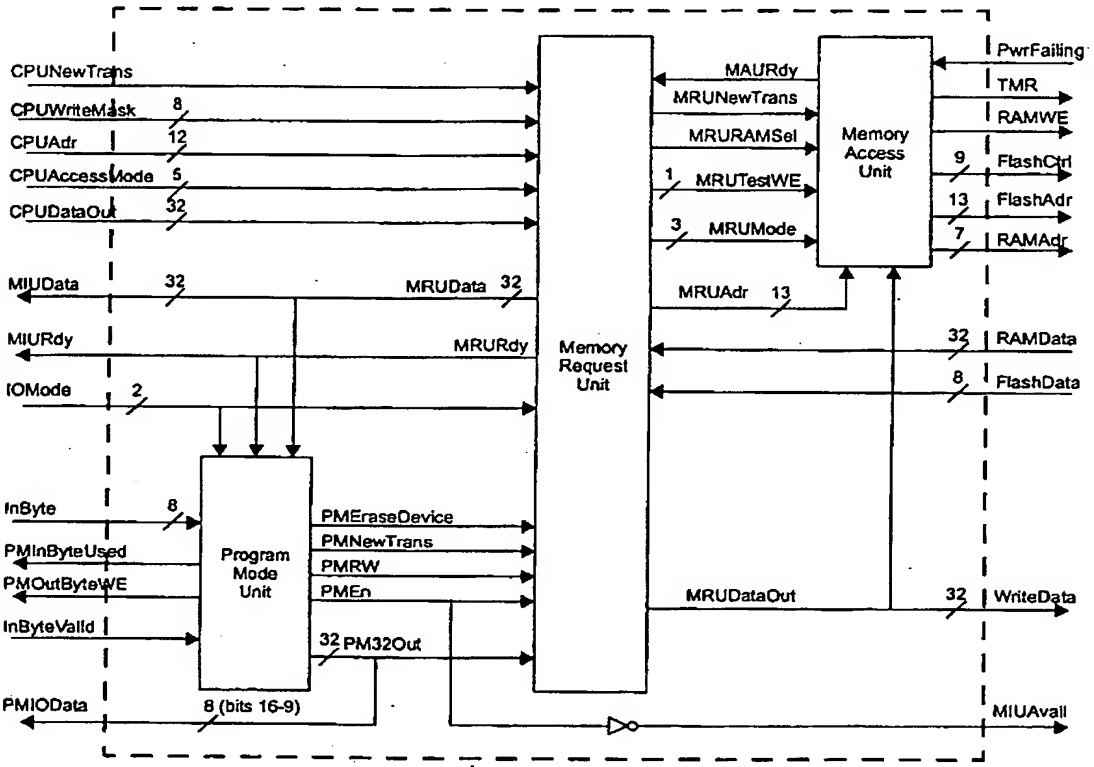


Figure 10. Block diagram of MIU

11.9 MEMORY COMPONENTS

The Memory Components block isolates the memory implementation from the rest of the QA Chip.

The entire contents of the Memory Components block must be protected from tampering. Therefore the logic must be covered by both Tamper Detection Lines. This is to ensure that program code, keys, and intermediate data values cannot be changed by an attacker. The 8-bit wide RAM also needs to be parity-checked.

Figure 11 shows a high level view of the Memory Components block. It consists of 8KBytes of flash memory and 3072 bits of parity checked RAM.

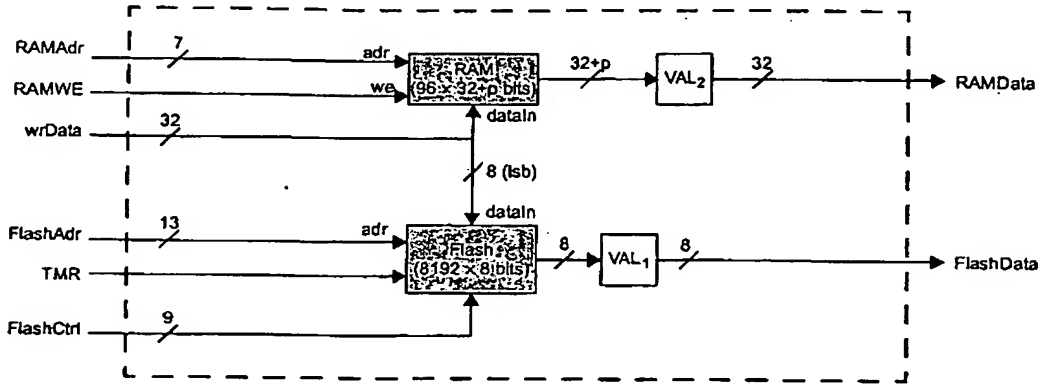


Figure 11. Block diagram of memory components

11.9.1 RAM

The RAM block is shown here as a simple 96 × 32-bit RAM (plus parity included for verification). The parity bit is generated during the write.

The RAM is in an unknown state after RESET, so program code cannot rely on RAM being 0 at startup.

The actual RAM will be 384 × 8-bits or 192 × 16 bits, based on library vendor choice and speed requirements. Speed requirements will be ascertained after simulation of program code, and the RAM substituted. In the meantime, 8-bit and 16-bit RAM can be readily simulated by making the RAM take 4 cycles and 2 cycles respectively to return a value (instead of 1 cycle).

11.9.2 Flash memory

A single Flash memory block is used to hold all non-volatile data. This includes program code and variables. The Flash memory block is implemented by TSMC component SFC0008_08B9_HE [4], which has the following characteristics:

- 8K × 8-bit arrangement
- 512 byte page erase
- Endurance of 20,000 cycles (min)
- Greater than 100 years data retention at room temperature
- Access time: 20 ns (max)
- Byte write time: 20µs (min)
- Page erase time: 20ms (min)
- Device erase time: 200 ms (min)
- Area of 0.494mm^2 (724.66µm × 682.05µm)

The FlashCtrl line are the various inputs on the SFC0008_08B9_HE required to read and write bytes, erase pages and erase the device. A total of 9 bits are required (see [4] for more information).

Flash values are unchanged by a RESET. After manufacture, the Flash contents must be considered to be garbage. After an erasure, the Flash contents in the SFC0008_08B9_HE is all 1s.

11.9.3 VAL blocks

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 35), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL_1, the effective byte output from the flash will always be 0 if the chip has been tampered with. This will cause shadow tests to fail, program code will not execute, and the chip will hang.

In the case of VAL_2, the effective byte from RAM will always be 0 if the chip has been tampered with, thus resulting in no temporary storage for use by an attacker.

12 I/O Unit

The I/O Unit (IOU) is responsible for providing the physical implementation of the logical interface described in Section 5.1 on page 5, moving between the various modes (Idle, Program, Trim and Active) according to commands sent by the master.

The IOU therefore contains the circuitry for communicating externally with the external world via the SClk and SDa pins. The IOU sends and receives data in 8-bit chunks. Data is sent serially, most significant bit (bit 7) first through to least significant bit (bit 0) last. When a master sends a command to an QA Chip, the command commences with a single byte containing an id in bits 7-2, an even parity bit for the id in bit 1, and a read/write sense in bit 0, as shown in Figure 12.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
PrilD5	PrilD4	PrilD3	PrilD2	PrilD1	PrilD0	Even_Par	R/~W

Figure 12. First byte sent to IOU

The IOU recognizes a global id of 0x00 and a local id (set after the CPU has executed program code due to a global id / ActiveMode command on the serial bus). Subsequent bytes contain modal information in the case of global id, and command/data bytes in the case of a match with the local id.

If the master sends data too fast, then the IOU will miss data, since the IOU never holds the bus. The meaning of too fast depends on what is running. In Program Mode, the master must send data a little slower than the time it takes to write the byte to flash (actually written as 2 8-bit writes, or 40µs). In Active Mode, the master can send data at rates up to 500 KHz.

None of the latches in the IOU need to be parity checked since there is no advantage for an attacker to destroy or modify them.

The IOU outputs 0s and inputs 0s if either of the Tamper Detection Lines is broken. This will only come into effect if an attacker has disabled the RESET and/or erase circuitry, since breaking either Tamper Detection Lines should result in a RESET or the erasure of all Flash memory.

The IOU's InByte, InByteValid, OutByte, and OutByteValid registers are used for communication between the master and the QA Chip. InByte and InByteValid provide the means for clients to pass commands and data to the QA Chip. OutByte and OutByteValid provide the means for the master to read data from the QA Chip.

- Reads from InByte should wait until InByteValid is set. InByteValid will remain clear until the master has written the next input byte to the QA Chip. When the IOU is told (by the FEU or MU) that InByte has been read, the IOU clears the InByteValid bit to allow the next byte to be read from the client.
- Writes to OutByte should wait until OutByteValid is clear. Writing OutByte sets the OutByteValid bit to signify that data is available to be transmitted to the master. OutByteValid will then remain set until the master has read the data from OutByte. If the master requests a byte but OutByteValid is clear, the IOU sends a NAck to indicate the data is not yet ready.

When the chip is reset via RstL, InByteValid and OutByteValid are both cleared and the IOU unit enters Idle Mode. The master is then able to send commands to the QA Chip as described in Section 5.1 on page 5.

The IOU re-initializes (and returns to Idle Mode) if it receives 3 stop bits in a row. This has a side effect of halting any currently executing program code in the CPU.

Figure 13 shows a block diagram of the IOU.

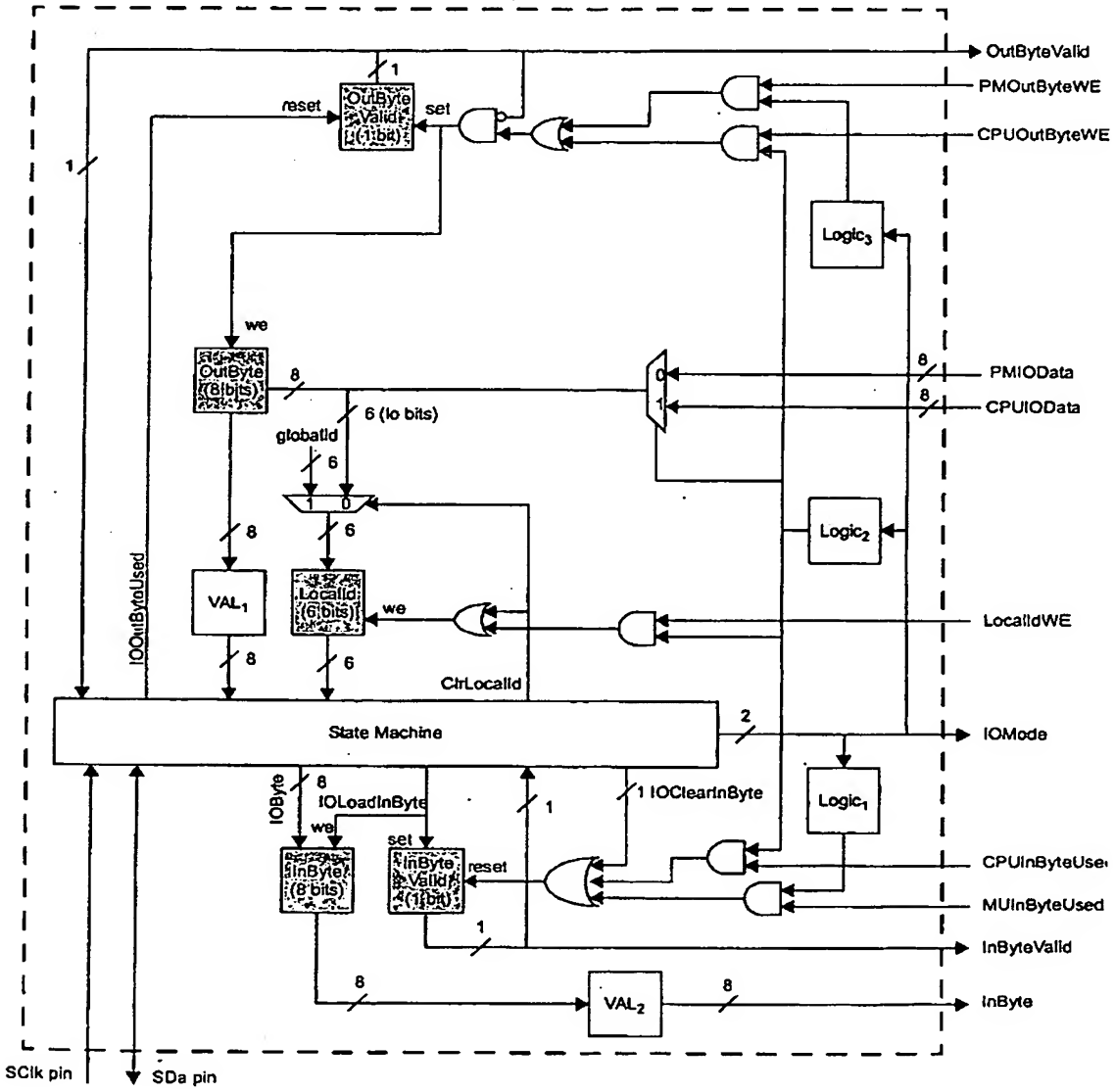


Figure 13. Block diagram of IOU

With regards to InByteValid inputs, set has priority over reset, although both set and reset in correct operation should never be asserted at the same time.

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in the Architecture Overview chapter), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL_1, the effective byte output from the chip will always be 0 if the chip has been tampered with. Thus no useful output can be generated by an attacker. In the case of VAL_2, the effective byte input to the chip will always be 0 if the chip has been tampered with. Thus no useful input can be chosen by an attacker.

There is no need to verify the registers in the IOU since an attacker does not gain anything by destroying or modifying them.

The current mode of the IOU is output as a 2-bit IOMode to allow the other units within the QA Chip to take correct action. IOMode is defined as shown in Table 26:

Table 26. IOMode values

Value	Interpretation
00	Idle Mode
01	Program Mode
10	Active Mode
11	Trim Mode

The Logic blocks generate a 1 if the current IOMode is in Program Mode, Active Mode or Trim Mode respectively. The logic blocks are:

$Logic_1$	IOMode = 01 (Program)
$Logic_2$	IOMode = 10 (Active)
$Logic_3$	IOMode = 11 (Trim)

12.1 STATE MACHINE

There are two state machines in the IOU running in parallel. The first is a byte-oriented state machine, the second is a bit-oriented state machine. The byte-oriented state machine keeps track of the operating mode of the QA Chip while the bit-oriented state machine keeps track of the low-level bit Rx/Tx protocol.

The SDa and SClk lines are connected to the respective pads on the QA Chip. The IOU passes each of the signals from the pads through 2 D-types to compensate for metastability on input, and then a further latch and comparitor to ensure that signals are only used if stable for 2 consecutive internal clock cycles. The circuit is shown in Section 12.1.1 below.

12.1.1 Start/Stop control signals

The StartDetected and StopDetected control signals are generated based upon monitoring SDa synchronized to SClk. The StartDetected condition is asserted on the falling edge of SDa synchronized to SClk, and the StopDetected condition is asserted on the rising edge of SDa synchronized to SClk.

In addition we generate feSClk which is asserted on the falling edge of SClk and reSClk which is asserted on the rising edge of SClk.

Figure 14 shows the relationship of inputs and the generation of reSClk, feSClk, StartDetected and StopDetected.

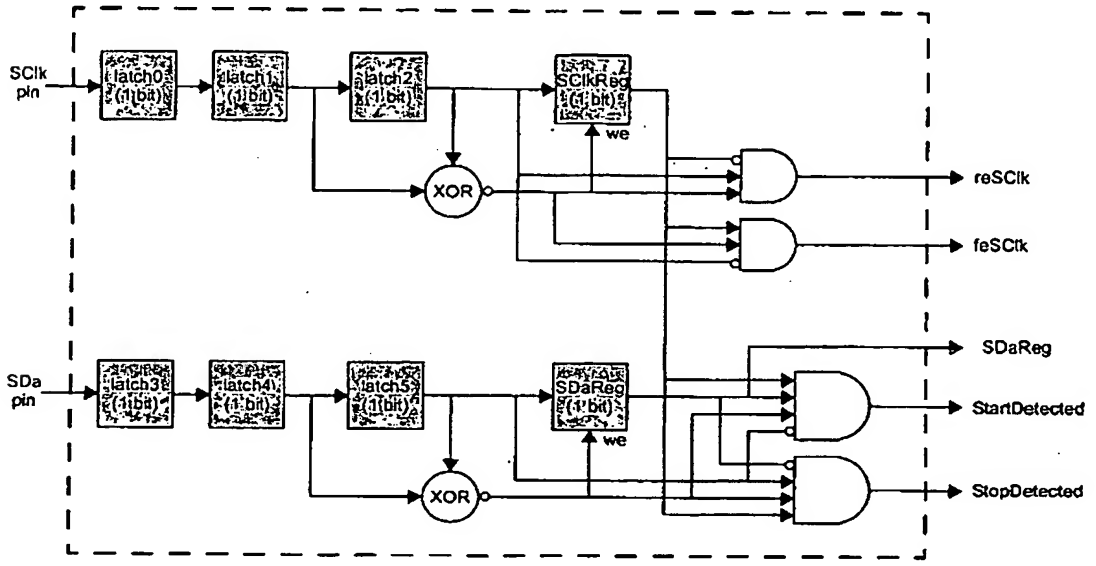


Figure 14. Relationship between external SDa and SClk and generation of internal signals

12.1.2 Control of SDa and SClk pins

The SClk line is always driven by the master. The SDa line is driven low whenever we want to transmit an ACK (SDa is active low) or a 0-bit from OutByte. The generation of the SDa pin is shown in the following pseudocode:

```
TxAck = (bitSM_state = ack) ∧
   ((byteSM_state = doWrite) ∨ (byteSM_state = getGlobalCmd) ∨
      ((byteSM_state = checkId) ∧ AckCmd))
TxBit ← (byteSM_state = doRead) ∧ (bitSM_state = xferBit) ∧ ¬OutByte¬bitCount
SDa = ¬(TxAck ∨ TxBit)# only drive the line when we are xmitting a 0
```

12.1.3 Bit-oriented state machine

The bit-oriented state machine keeps track of the general flow of serial transmission including start/data/ack/stop as shown in the following pseudocode:

```
idle
   EndByte = FALSE
   EndAck = FALSE
   If (StartDetected)
      state ← starting
   Else
      state ← idle
   EndIf

starting
   EndByte = FALSE
   EndAck = FALSE
   NAck ← 0
   If (StopDetected)
      state ← idle
   ElseIf (feSClk)# ignore the falling edge of the serial clock
      bitCount ← 0
      state ← xferBit
   Else
      state ← starting# includes StartDetected
   EndIf

xferBit
   EndAck = FALSE
   EndByte = (feSclk ∧ (bitCount = 7))
   If (reSClk)
      shiftLeft[ioByte, SDaReg]# capture the bit in the ioByte shift register
   EndIf
   If (feSClk)
      bitCount ← bitCount + 1# modulo count due to 3 bit bitCount
   EndIf
   If (StopDetected)
      state ← idle
   ElseIf (StartDetected)
      state ← starting
   ElseIf (EndByte)
      state ← ack
   Else
      state ← xferBit
   EndIf

ack
   EndByte = FALSE
   EndAck = feSclk
   If (StopDetected)
      state ← idle
   ElseIf (StartDetected)
      state ← starting
   ElseIf (EndAck)
      state ← xferBit# bitCount is already 0
   Else
      If (reSClk)
         NAck ← SDaReg# active low, so 0 = ACK, 1 = NACK
      EndIf
      state ← ack
   EndIf
```

12.1.4 Byte-oriented state machine

The following pseudocode illustrates the general startup state of the IOU and the receipt of a transmission from the master.

```
init  # entry on RstL
   IOMode ← IdleMode
   ClearLocalId = 1# loads localId with the global Id
   state ← idle

idle
   If (StartDetected)
      state ← checkId
   Else
      state ← idle
   EndIf

stop1
   If (StopDetected)
      state = stop2
   ElseIf (StartDetected)
      state = checkId
   Else
      state ← stop1
   EndIf

stop2
   If (StopDetected)
      state = init            # prepares for possible power down
   ElseIf (StartDetected)
      state = checkId
   Else
      state ← stop2
   EndIf

checkId
   globalW = (ioByte7-0 = globalID+parity+W)
   localW = (ioByte7-0 = localID+parity+W)
   localR = (ioByte7-0 = localID+parity+R)
   If (StopDetected)
      state ← stop1
   ElseIf (EndByte)
      AckCmd ← (globalW ∨ localW ∨ (localR ∧ OutByteValid))
   ElseIf (EndAck)
      If (globalW)# global has precedence over local
         IOMode ← IdleMode# jic - any output was pending
         IOOutByteUsed = 1
         IOClearInByte = 1# ensure there is nothing hanging around from before
         state ← getGlobalCmd
      ElseIf (localW)
         IOMode ← IdleMode# jic - any output was pending
         IOOutByteUsed = 1
         IOClearInByte = 1# ensure there is nothing hanging around from before
         state ← getLocalCmd
      ElseIf (localR ∧ IOMode1 ∧ AckCmd)
         state ← doRead
      Else
         state ← idle# ignore reads unless first in active or trim mode
      EndIf
   Else
      state ← checkId
   EndIf
```

With a new global command the IOU acknowledges receipt of the global id and waits for the mode byte (see Table 2) to determine the new operating mode:

```
getGlobalCmd
   If (StopDetected)
      state ← stop1
   ElseIf (StartDetected)
      state ← checkId
   ElseIf (EndAck)
      If (ioByte = ProgramModeId)
         IOMode ← ProgramMode
         state ← doWrite
      ElseIf (ioByte = ActiveModeId)
         IOMode ← ActiveMode
         IOLoadInByte = 1 # transfers ActiveModeId to InByte
         state ← doWrite
      ElseIf (ioByte = TrimModeId) # don't loadInByte, so ignore ActiveMode byte
         IOMode ← TrimMode
         state ← doWrite
      Else
         state ← idle# unknown id, so ignore remainder
      EndIf
   Else
      state ← getGlobalCmd
   EndIf
```

When the master sends a new local command, the opcode byte must be in a specific format. If it is in the correct format, the mode changes to Active and the command is executed, as shown in the following pseudocode:

```
getLocalCmd
   If (StopDetected)
      state ← stop1
   ElseIf (StartDetected)
      state ← checkId
   ElseIf (EndByte)
      If ((ioByte5-3 = ¬ioByte2-0) ∧ (ioByte7-6 = countSetBits[ioByte2-0]))
         IOMode ← ActiveMode
         IOLoadInByte = 1 # transfers ioByte to InByte
         state ← doWrite
      Else
         state ← idle# bad command, so ignore remainder
      EndIf
   Else
      state ← getLocalCmd
   EndIf
```

When the master writes bytes to the QA Chip (parameters for a command), the program must consume the byte fast enough (i.e. during the sending of the ACK) or subsequent bits may be lost. The process of receiving bytes is shown in the following pseudocode:

```
doWrite
   If (StopDetected)# from bit transfer state machine
      state ← stopl  # stay in whatever IOMode we were in
   ElseIf (StartDetected)
      state ← checkId
   Else
      If (EndByte)# from bit transfer state machine (on falling edge of srClk)
         IOLoadInByte = ¬InByteValid
      EndIf
      If (EndByte ∧ InByteValid)# will only be when master sends data too quickly
            state ← idle       . # ACK will not be sent when in idle state
      Else
         state ← doWrite # ACK will be sent automatically after byte is Rxed
      EndIf
   EndIf
```

When the master wants to read, the IOU sends one byte at a time as requested. The process is shown in the following pseudocode:

```
doRead
   If (StopDetected)
      state ← stopl
   ElseIf (StartDetected)
      state ← checkId
   ElseIf (EndAck)
      If (NAck ∨ ¬OutByteValid)
         state ← idle
      Else
         state ← doRead
      EndIf
   Else
      If (EndByte)
         IOOutByteUsed = 1
      EndIf
      state ← doRead
   EndIf
```

13 Fetch and Execute Unit

13.1 INTRODUCTION

The QA Chip does not require the high speeds and throughput of a general purpose CPU. It must operate fast enough to perform the authentication protocols, but not faster. Rather than have specialized circuitry for optimizing branch control or executing opcodes while fetching the next one (and all the complexity associated with that), the state machine adopts a simplistic view of the world. This helps to minimize design time as well as reducing the possibility of error in implementation.

The FEU is responsible for generating the operating cycles of the CPU, stalling appropriately during long command operations due to memory latency.

When a new transaction begins, the FEU will generate a RST (reset) instruction.

The general operation of the FEU is to generate sets of cycles:

- Cycle 0: *fetch cycles*. This is where the opcode is fetched from the program memory, and the effective address from the fetched opcode is generated. The Fetch output flag is set during the final cycle 0 (i.e. when the opcode is finally valid).
- Cycle 1: *execute cycle*. This is where the operand is (potentially) looked up via the generated effective address (from Cycle 0) and the operation itself is executed. The Exec output flag is set during the final cycle 1 (i.e. when the operand is finally valid).

Under normal conditions, the state machine generates multiple Cycle=0 followed by multiple Cycle=1. This is because the program is stored in flash memory, and may take multiple cycles to read. In addition, writes to and erasures of flash memory take differing numbers of cycles to perform. The FEU will stall, generating multiple instances of the same Cycle value with Fetch and Exec both 0 until the input MIURdy = 1, whereupon a Fetch or Exec pulse will be generated in that same cycle.

There are also two cases for stalling due to serial I/O operations:

- The opcode is ROR OutByte, and OutByteValid = 1. This means that the current operation requires outputting a byte to the master, but the master hasn't read the last byte yet.
- The operation is ROR InByte, and InByteValid = 0. This means that the current operation requires reading a byte from the master, but the master hasn't supplied the byte yet.

In both these cases, the FEU must stall until the stalling condition has finished.

Finally, the FEU must stop executing code if the IOU exits Active Mode.

The local Cmd opcode/operand latch needs to be parity-checked. The logic and registers contained in the FEU must be covered by both Tamper Detection Lines. This is to ensure that the instructions to be executed are not changed by an attacker.

13.2 STATE MACHINE

The Fetch and Execute Unit (FEU) is combinatorial logic with the following registers:

Table 27. FEU Registers

Name	#bits	Description
Output registers (visible outside the FEU)		
Cycle	1	0 if the FEU is currently fetching an opcode, 1 if the FEU is currently executing the opcode.
newMemTrans	1	Is asserted during the start of a potential new memory access. 0 = this is not the first cycle of a set of Cycle 0 or Cycle 1 1 = this is the first cycle of a set of Cycle 0 or Cycle 1 (previous cycle must have been a Fetch or an Exec).
Local registers (not visible outside the FEU)		
Go	1	1 if the FEU is currently executing program code, 0 if it is not.
currCmd	8+p	Holds the currently executing instruction (parity checked).
pendingKill	1	The currently executing program is waiting to be halted (waiting due to memory access)
pendingStart	1	A new transaction is waiting to be started (waiting due to memory access or an existing transaction not yet stopped)
wasIdle	1	The previous cycle had an IOMode of IdleMode.

In addition, the following externally visible outputs are generated asynchronously:

Table 28. Externally visible asynchronous FEU outputs

Name	#bits	Description
Fetch	1	1 if the FEU is performing the final cycle of a fetch (i.e. Cycle will also be 0). It is set when the NextCmd output is valid. The local Cmd register is latched during the Fetch cycle with either the incoming MIU8Data or an FEU-generated command.
Exec	1	1 if the FEU is performing the final cycle of an execute (i.e. Cycle will also be 1). It is set when the data required by the opcode from the MIU is valid. Other units can execute the Cmd and latch data from the MIU (e.g. from MIUData) during the Exec cycle.
Cmd	8	When Cycle = 0, this holds the next instruction to be executed (during the next Cycle = 1). Is generated based on incoming MIU8Data or substituted FEU command (e.g. JSR 0). When Cycle = 1, this holds the current instruction being executed (based on theCmd).

The Cycle and currCmd registers are not used directly. Instead, their outputs are passed through a VAL unit before use. The VAL units are designed to validate the data that passes through them. Each contains an OK bit connected to both Tamper Prevention and Detection Lines. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL_1, the effective Cycle will always be 0 if the chip has been tampered with. Thus no program code will execute.

In the case of VAL_2, the effective 8-bit currCmd value will always be 0 if the chip has been tampered with. Multiple 0s will be interpreted as the JSR 0 instruction, and this will effectively hang the CPU. VAL_2 also performs a parity check on the bits from currCmd to ensure

that currCmd has not been tampered with. If the parity check fails, the Erase Tamper Detection Line is triggered. For more information on Tamper Prevention and Detection circuitry, see Section 10.3.5 on page 35.

13.2.1 Pseudocode

```
reset
   Fetch = 0
   Exec = 0
   Cycle ← 0
   currCmd ← 0
   Go ← 0
   pendingKill ← 0
   pendingStart ← 0
   newMemTrans ← 0
   wasIdle ← 0

main
   isActive = (IOMode = ActiveMode)
   wasIdle ← (IOMode = IdleMode)
   wantToStart = (pendingStart ∨ wasIdle) ∧ isActive
   newTrans = wantToStart ∧ ¬Go ∧ MIUAvail
   pendingStart ← wantToStart ∧ ¬newTrans
   killTrans = Go ∧ (¬isActive ∨ pendingKill)

   Fetch = newTrans ∨ (Go ∧ ¬Cycle ∧ MIURdy ∧ ¬killTrans)
   inDelay = (currCmd = ROR InByte) ∧ ¬InByteValid
   outDelay = (currCmd = ROR OutByte) ∧ OutByteValid
   ioDelay = inDelay ∨ outDelay
   Exec = Go ∧ Cycle ∧ MIURdy ∧ ¬ioDelay

   If (Cycle)
      Cmd = currCmd
   ElseIf (newTrans)
      Cmd = RST# reset
   Else
      Cmd = MIU8Data
   EndIf

   resetGo = (MIURdy ∧ killTrans) ∨ (Fetch ∧ (Cmd = HALT))
   pendingKill ← killTrans ∧ ¬resetGo

   changeCycle = Fetch ∨ Exec# will only be 1 when Go = 1
   Cycle ← newTrans ∨ ((Cycle ⊕ changeCycle) ∧ ¬resetGo)
   newMemTrans ← newTrans ∨ (changeCycle ∧ ¬resetGo)
   If (Fetch)
      currCmd ← Cmd
   EndIf

   If (resetGo)
      Go ← 0
   ElseIf (newTrans)
      Go ← 1
   EndIf
```

14 ALU

The Arithmetic Logic Unit (ALU) contains a 32-bit Acc (Accumulator) register as well as the circuitry for simple arithmetic and logical operations.

The logic and registers contained in the ALU must be covered by both Tamper Detection Lines. This is to ensure that keys and intermediate calculation values cannot be changed by an attacker. In addition, the Accumulator must be parity-checked.

A 1-bit Z register contains the state of zero-ness of the Accumulator. The Accumulator is cleared to 0 upon a RstL, and the Z register is set to 1. The Accumulator is updated for any of the commands: AND, OR, XOR, ADD, ROR, and RIA, and the Z register is updated whenever the Accumulator is updated.

Each arithmetic and logical block operates on two 32-bit inputs: the current value of the Accumulator, and the current 32-bit output of the DataSel block (either the 32 bit value from MIUData or an immediate value). The AND, OR, XOR and ADD blocks perform the standard 32-bit operations. The remaining blocks are outlined below.

Figure 15 shows a block diagram of the ALU:

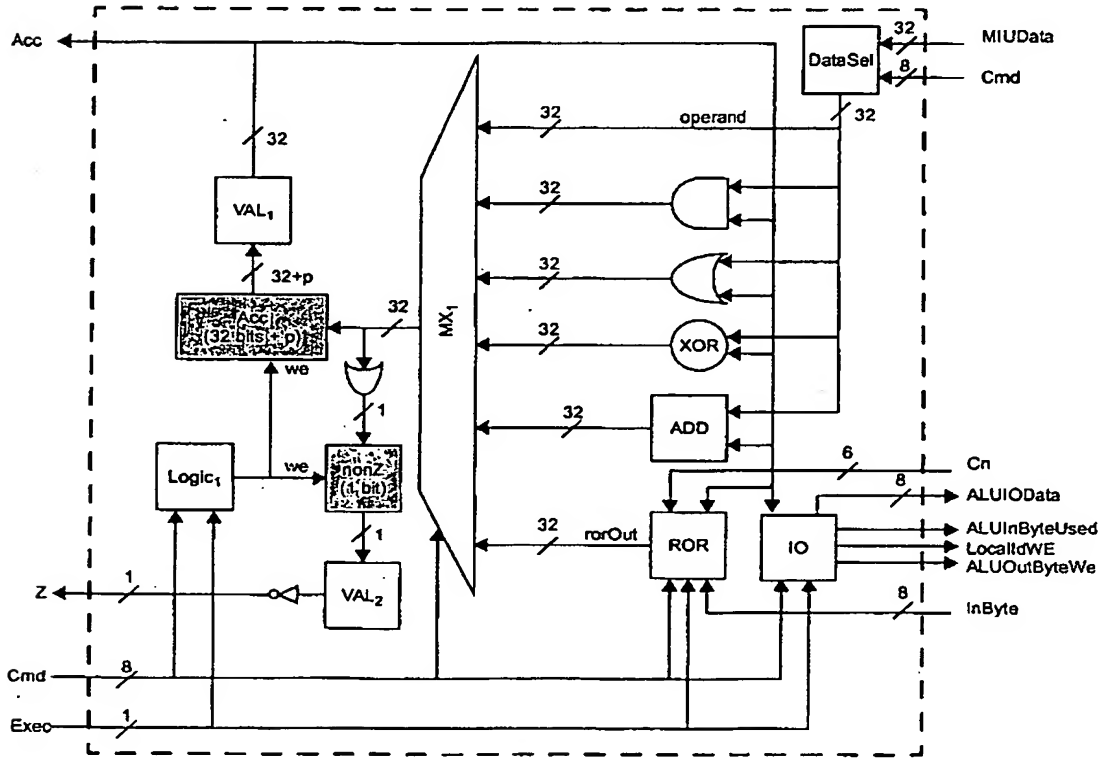


Figure 15. Block diagram of ALU

The Accumulator is updated for all instructions where the high bit of the opcode is set:

$Logic_1$	Exec ∧ Cmd_7

Since the WriteEnables of Acc and Z takes Cmd_7 and Exec into account (due to $Logic_1$), these two bits are not required by the multiplexor MX_1 in order to select the output. The output selection for MX_1 only requires bits 6-3 of the Cmd and is therefore simpler as a result (as shown in Table 29).

Table 29. Selection for multiplexor MX_1

	Output	Cmd[7:0]
MX_1	ImmOut	011x v 1110 (LD)
	rorOut	100x v 1111 (RIA, ROR)
	from XOR	001x v 1100 (XOR)
	from ADD	010x v 1101 (ADD)
	from AND	0000 v 1010 (AND)
	from OR	0001 v 1011 (OR)

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 35), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit.

In the case of VAL_1, the effective bit output from the Accumulator will always be 0 if the chip has been tampered with. This prevents an attacker from processing anything involving the Accumulator. VAL_1 also performs a parity check on the Accumulator, setting the Erase Tamper Detection Line if the check fails.

In the case of VAL_2, the effective Z status of the Accumulator will always be true if the chip has been tampered with. Thus no looping constructs can be created by an attacker.

14.1 DataSel Block

The DataSel block is designed to implement the selection between the MIU32Data and the immediate addressing mode for logical commands.

Immediate addressing relies on 3 bits of operand, plus an optional 8 bits at PC+1 to determine an 8-bit base value. Bits 0 to 1 determine whether the base value comes from the opcode byte itself, or from PC+1, as shown in Table 30.

Table 30. Selection for base value in immediate mode

Cmd_{1-0}	Base value
00	00000000
01	00000001
10	From PC+1 (i.e. $MIUData_{31-24}$)
11	11111111

The base value is computed by using CMD_0 as bit 0, and copying CMD_1 into the upper 7 bits.

The 8-bit base value forms the lower 8 bits of output. These 8 bits are also ANDed with the sense of whether the data is replicated in the upper bits or not (i.e. CMD_2). The resultant bits are copied in 3 times to form the upper 24 bits of the output.

Figure 16 shows a block diagram of the ALU's DataSel block:

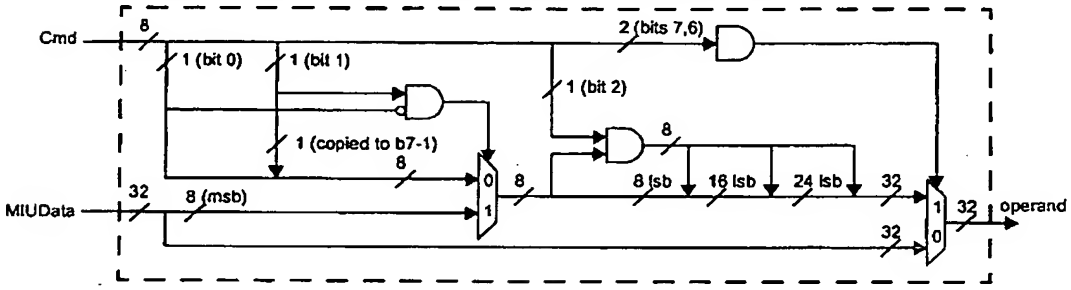


Figure 16. Block diagram of DataSel

14.2 ROR Block

The ROR block implements the ROR and RIA functionality of the ALU.

A 1-bit register named RTMP is contained within the ROR unit. RTMP is cleared to 0 on a RstL, and set during the ROR RB and ROR XRB commands. The RTMP register allows implementation of Linear Feedback Shift Registers with any tap configuration.

Figure 17 shows a block diagram of the ALU's ROR block:

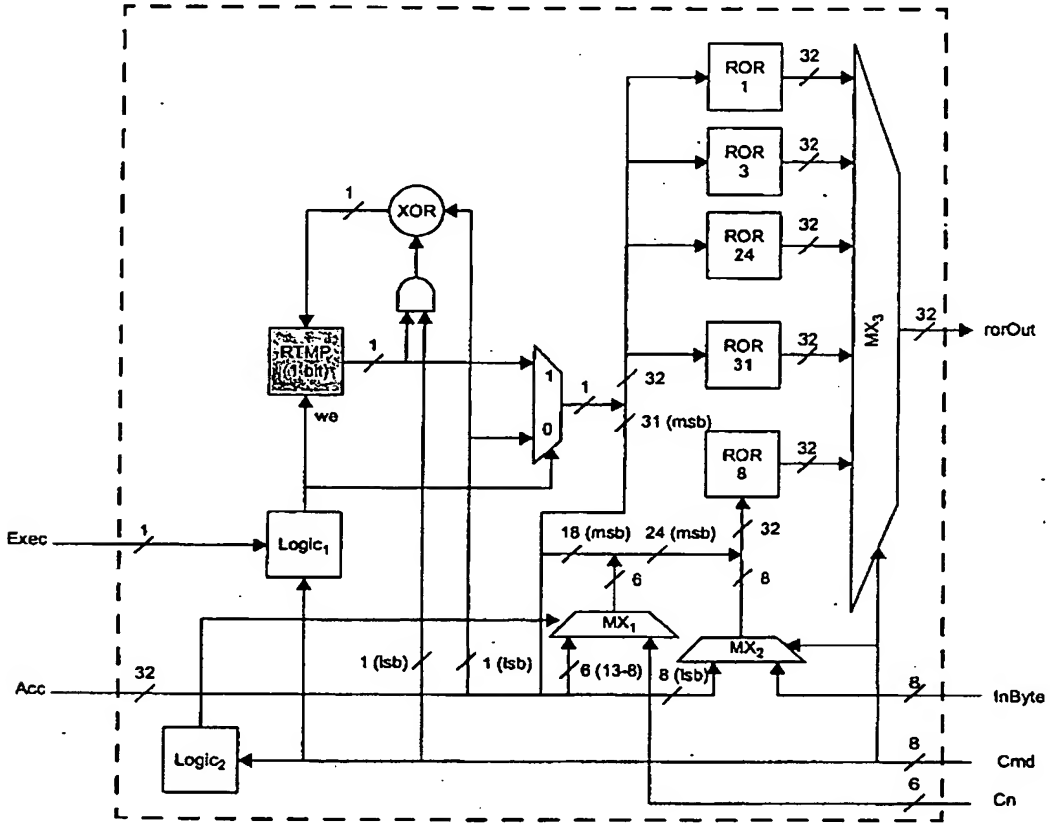


Figure 17. Block diagram of ROR

The ROR n, blocks are shown for clarity, but in fact would be hardwired into multiplexor MX_3, since each block is simply a rewiring of the 32-bits, rotated right n bits.

$Logic_1$ is used to provide the WriteEnable signal to RTMP. The RTMP register should only be written to during ROR RB and ROR XRB commands. The combinatorial logic block is:

$Logic_1$	$Exec \wedge (Cmd_{7-4} = ROR) \wedge (Cmd_{3-1} = 000)$

Multiplexor MX_1 performs the task of selecting the 6-bit value from Cn instead of bits 13-8 (6 bits) from Acc (the selection is based on the value of $Logic_2$). Bit 5 is required to distinguish ROR from RIA.

$Logic_2$	Cmd_{5-2} = 0x10

Table 31. Selection for multiplexor MX_1

	Output	Logic
MX_1	Cn	1
	Acc_{13-8}	0

Multiplexor MX_2 performs the task of selecting the 8-bit value from InByte instead of the lower 8 bits from the ANDed Acc based on the CMD.

Table 32. Selection for multiplexor MX_2

	Output	Cmd
MX_2	InByte	0x110
	Acc_{7-0}	¬(0x110)

Multiplexor MX_3 does the final rotating of the 32-bit value. The bit patterns of the CMD operand are taken advantage of:

Table 33. Selection for multiplexor MX_3

	Output	Cmd	Comments
MX_3	ROR 1	00xx	RB, XRB, WriteMask, 1
	ROR 3	010x	3
	ROR 31	0110	31
	ROR 24	0111	24
	ROR 8	1xxx	RIA, InByte, 8, OutByte, C1, C2, ID

14.3 IO Block

The IO block within the ALU implements the logic for communicating with the IOU during instructions that involve the Accumulator. This includes generating appropriate control signals and for generating the correct data for sending during writes to the IOU's OutByte and LocalId registers.

Figure 18 shows a block diagram of the IO block:

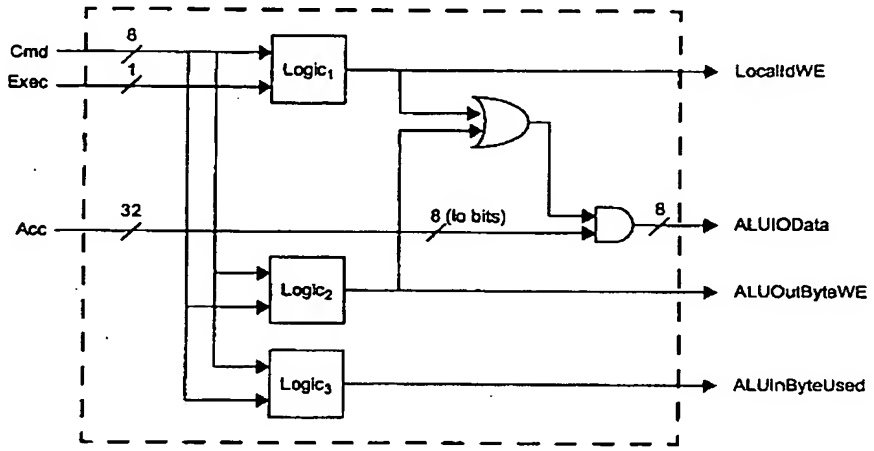


Figure 18. Block diagram of the ALU's IO block

$Logic_1$ is used to provide the LocalIdWE signal to the IOU. The localId register should only be written to during the ROR ID command. Only the lower 6 bits of the Accumulator are written to the localId register.

$Logic_2$ is used to provide the ALUOutByteWE signal to the IOU. The OutByte register should only be written to during the ROR OutByte command. Only the lower 8 bits of the Accumulator are written to the OutByte register.

In both cases we output the lower 8 bits of the Accumulator. The ALUIOData value is ANDed with the output of $Logic_2$ to ensure that ALUIOData is only valid when it is safe to do so (thus the IOU logic never sees the key passing by in ALUIOData). The combinatorial logic blocks are:

$Logic_1$	Exec ∧ (Cmd_{7-0} = ROR ID)
$Logic_2$	Exec ∧ (Cmd_{7-0} = ROR OutByte)

$Logic_3$ is used to provide the ALUInByteUsed signal to the IOU. The InByte is only used during the ROR InByte command. The combinatorial logic is:

$Logic_3$	Exec ∧ (Cmd_{7-0} = ROR InByte)

15 Program Counter Unit

The Program Counter Unit (PCU) includes the 12 bit PC (Program Counter), as well as logic for branching and subroutine control.

The PCU latches need to be parity-checked. In addition, the logic and registers contained in the PCU must be covered by both Tamper Detection Lines to ensure that the PC cannot be changed by an attacker.

The PC is implemented as a 12 entry by 12-bit PCA (PC Array), indexed by a 4-bit SP (Stack Pointer) register. The PC, PCRamSel and SP registers are all cleared to 0 on a RstL, and updated during the flow of program control according to the opcodes.

The current value for the PC is normally updated during the Execute cycle according to the command being executed. However it is also incremented by 1 during the Fetch cycle for two byte instructions such as JMP, JSR, DBR, TBR, and instructions that require an additional byte for immediate addressing. The mechanism for calculating the new PC value depends upon the opcode being processed.

Figure 19 shows a block diagram of the PCU:

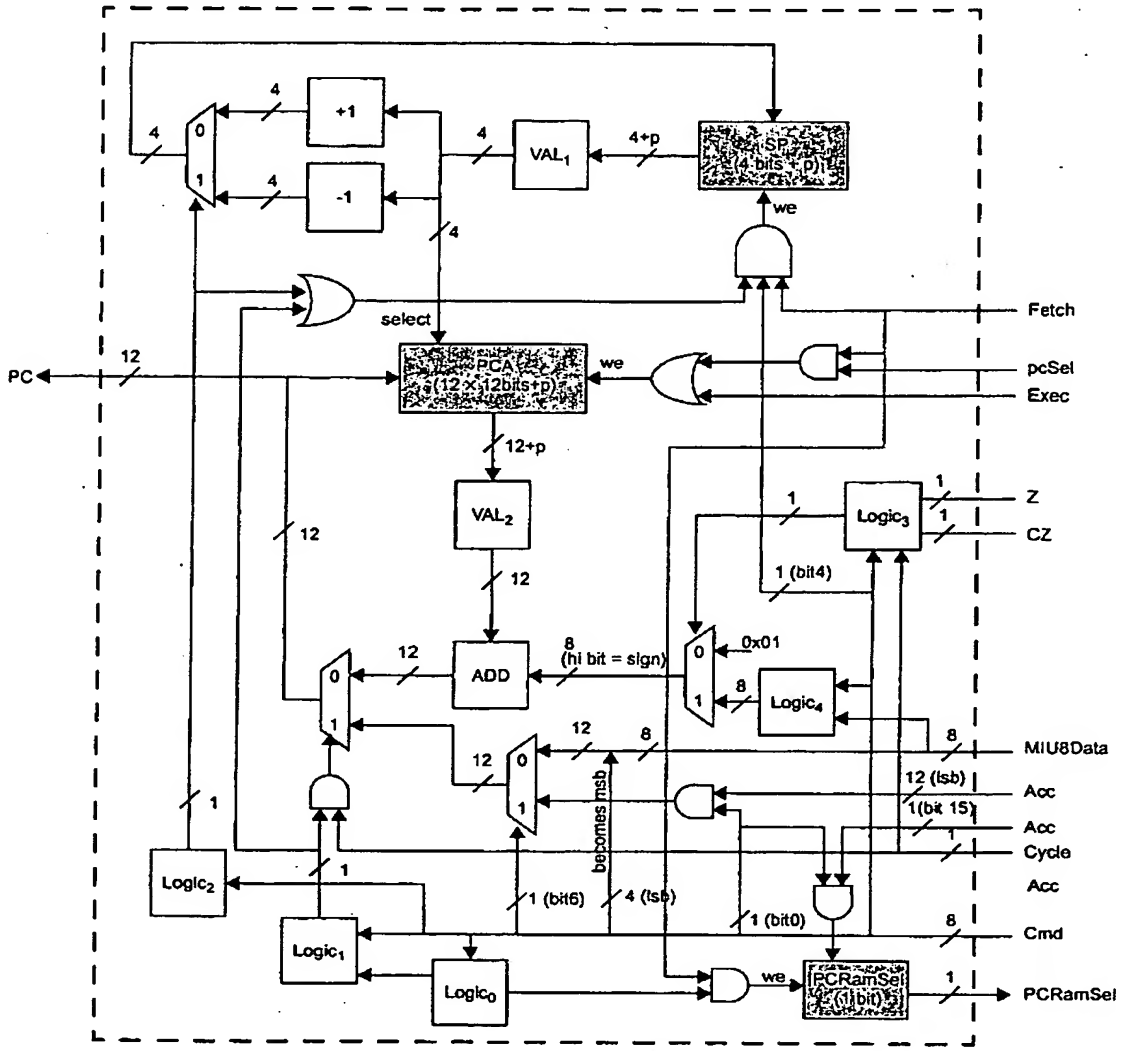


Figure 19. Block diagram of PCU

The ADD block is a simple adder modulo 2^{12} with two inputs: an unsigned 12 bit number and an 8-bit signed number (high bit = sign). The signed input is either a constant of 0x01, or an 8-bit offset (the 8 bits from the MIU).

The "+1" block takes a 4-bit input and increments it by 1 (modulo 12). The "-1" block takes a 4-bit input and decrements it by 1 (modulo 12).

Table 34 lists the different forms of PC control:

Table 34. Different forms of PC control during the Exec cycle

Command	Action
JMP	The PC is loaded with the current 12-bit value as passed in from the MIU.
JPI	The PC is loaded with the current 12-bit value as passed in from the Acc. PCRamSel is loaded with the value from bit 15 of the Acc.
RST	The PC is loaded with 0. PCRamSel is loaded with 0 (program in flash)
JSR, JSI	Save old value of PC onto stack for later. The PC is loaded with the current 12-bit value as passed in from either the MIU or the Acc. With JSI, PCRamSel is loaded from the value in bit 15 of the Accumulator.
RTS	Pop old value of PC from stack and increment by 1 to get new PC.
TBR	If the Z flag matches the TRB test, add 8-bit signed number (MIU8Data) to current PC. Otherwise increment current PC by 1.
DBR	If the CZ flag is set, add 8-bit signed offset (MIU8Data) to current PC. Otherwise increment current PC by 1.
All others	Increment current PC by 1

The updating of PCRamSelect only occurs during JPI, JSI and RST instructions, detected via $Logic_0$. The same action for the Exec takes place for JMP, JSR, JPI, JSI and RST, so we specifically detect that case in $Logic_1$. In the same way, we test for the RTS case in $Logic_2$.

$Logic_0$	$Cmd_{7-1} = 011x001$
$Logic_1$	$(Cmd_{7-5} = 000) \vee Logic_0$
$Logic_2$	$Cmd_{7-0} = RTS$

When updating the PC, we must decide if the PC is to be replaced by a completely new value (as in the case of the JMP, JSR, JPI and JSI instructions), or by the result of the adder (all other instructions). The output from $Logic_1$ ANDed with Cycle can therefore be safely used by the multiplexor to obtain the new PC value (we need to always select PC+1 when Cycle is 0, even though we don't always write it to the PCA).

Note that the RST instruction is implemented as 12 AND gates that cause the Accumulator value to be ignored, and the new PC to be set to 0. Likewise, the PCRamSelect bit is cleared via the RST instruction using the same AND mechanism.

The input to the 12-bit adder depends on whether we are incrementing by 1 (the usual case), or adding the offset as read from the MIU (when a branch is taken by the DBR and TBR instructions). $Logic_3$ generates the test.

$Logic_3$	$Cycle \wedge (((Cmd_{7-4} = DBR) \wedge \neg CZ) \vee ((Cmd_{7-4} = TBR) \wedge (Cmd_0 \oplus Z)))$

The actual offset to be added in the case of the DBR and TBR instructions is either the 8-bit value read from the MIU, or an 8-bit value generated by bits 3-1 of the opcode and treating bit 4 of the opcode as the sign (thereby making DBR immediate branching negative, and TBR immediate branching positive). The former is selected when bits 3-1 of the opcode is 0, as shown by $Logic_4$.

$Logic_4$	If $(Cmd_{3-1} = 000)$ output MIU8Data Else output $Cmd_4 \mid Cmd_4 \mid Cmd_4 \mid Cmd_4 \mid Cmd_4 \mid Cmd_{3-1}$

Finally, the selection of which PC entry to use depends on the current value for SP. As we enter a subroutine, the SP index value must increment, and as we return from a subroutine, the SP index value must decrement. $Logic_1$ tells us when a subroutine is being entered, and $Logic_2$ tells us when the subroutine is being returned from. We use $Logic_2$ to select the altered SP value, but only write to the SP register when Exec and Cmd_4 are also set (to prevent JMP and RST from adjusting SP).

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 35), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. Both VAL units also parity-check the data bits to ensure that they are valid. If the parity-check fails, the Erase Tamper Detection Line is triggered.

In the case of VAL_1, the effective output from the SP register will always be 0. If the chip has been tampered with. This prevents an attacker from executing any subroutines.

In the case of VAL_2, the effective PC output will always be 0 if the chip has been tampered with. This prevents an attacker from executing any program code.

16 Address Generator Unit

The Address Generator Unit (AGU) generates effective addresses for accessing the Memory Unit (MU). In Cycle 0, the PC is passed through to the MU in order to fetch the next opcode. The AGU interprets the returned opcode in order to generate the effective address for Cycle 1. In Cycle 1, the generated address is passed to the MU.

The logic and registers contained in the AGU must be covered by both Tamper Detection Lines. This is to ensure that an attacker cannot alter any generated address. The latches for the counters and calculated address should also be parity-checked.

If either of the Tamper Detection Lines is broken, the AGU will generate address 0 each cycle and all counters will be fixed at 0. This will only come into effect if an attacker has disabled the RESET and/or erase circuitry, since under normal circumstances, breaking a Tamper Detection Line will result in a RESET or the erasure of all Flash memory.

16.1 IMPLEMENTATION

The block diagram for the AGU is shown in Figure 20:

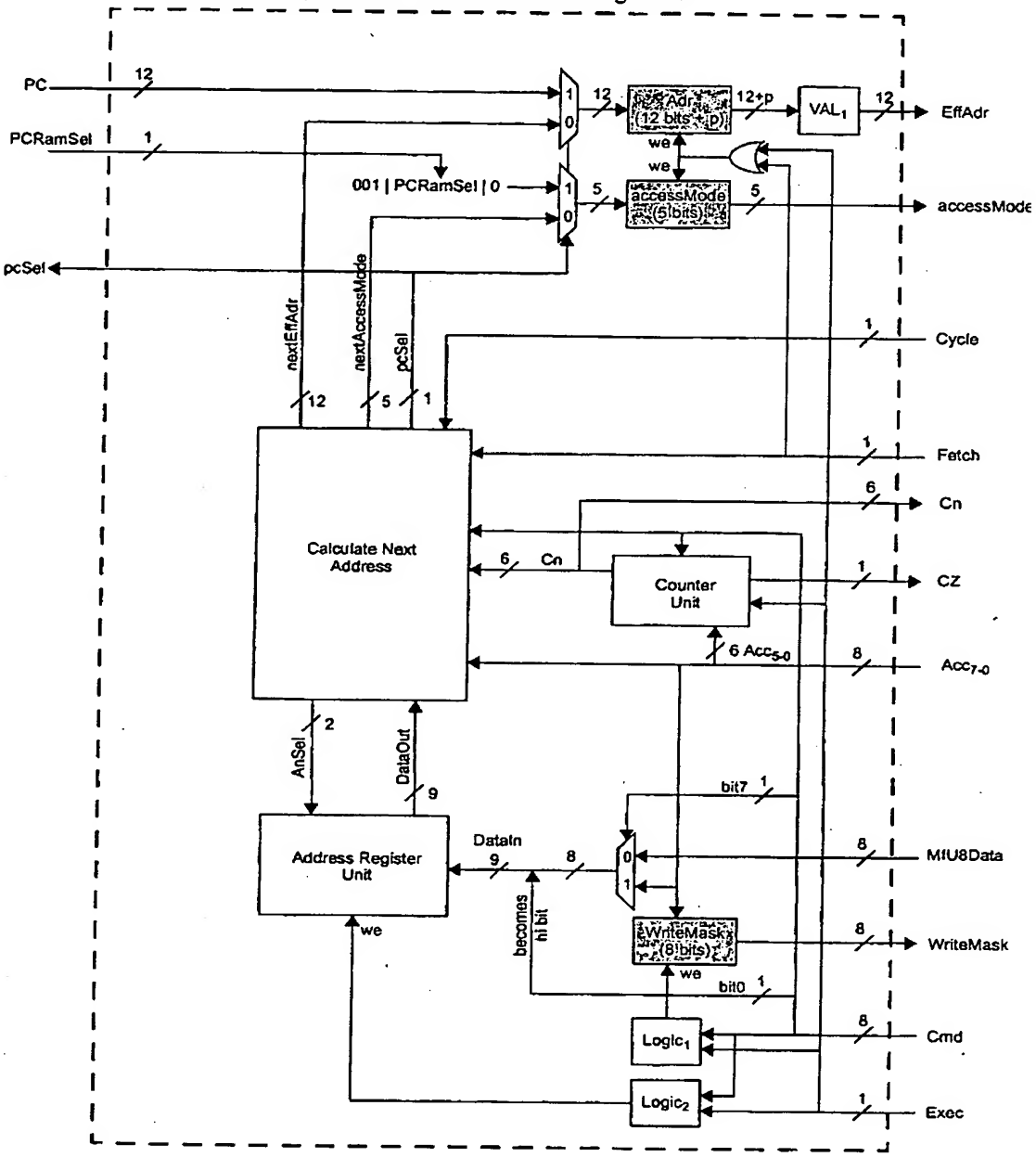


Figure 20. Block diagram of Address Generator Unit

The accessMode and WriteMask registers must be cleared to 0 on reset to ensure that no access to memory occurs at startup of the CPU.

The Adr and accessMode registers are written to during the final cycle of cycle 0 (Fetch) and cycle 1 (Exec) with the address to use during the following cycle phase. For example, when cycle = 1, the PC is selected so that it casn be written to Adr during Exec. During cycle 0, while the PC is being output from Adr, the address to be used in the following cycle 1 is calculated (based on the fetched opcode seen as Cmd) and finally stored in Adr when Fetch is 1. The accessMode register is also updated in the same way.

It is important to distinguish between the value of Cmd during different values for Cycle:

- During Cycle 0, when Fetch is 1, the 8-bit input Cmd holds the instruction to be executed in the following Cycle 1. This 8-bit value is used to decode the effective address for the operand of the instruction.
- During Cycle 1, when Exec is 1, Cmd holds the currently executing instruction.

The WriteMask register is only ever written to during execution of an appropriate ROR instruction. $Logic_1$ sets the WriteMask and MMR WriteEnables respectively based on this condition:

$Logic_1$	$Exec \wedge (Cmd_{7-0} = ROR\ WriteMask)$

The data written to the WriteMask register is the lower 8 bits of the Accumulator.

The Address Register Unit is only updated by an RIA or LIA instruction, so the writeEnable is generated by $Logic_2$ as follows:

$Logic_2$	$Exec \wedge (Cmd_{8-3} = 1111)$

The Counter Unit (CU) generates counters C1, C2 and the selected N index. In addition, the CU outputs a CZ flag for use by the PCU. The CU is described in more detail below.

The VAL_1 unit is a validation unit connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 35). It contains an OK bit that is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with the 12 bits of Adr before they can be used. If the chip has been tampered with, the address output will be always 0, thereby preventing an attacker from accessing other parts of memory. The VAL_1 unit also performs a parity check on the Adr Address bits to ensure it has not been tampered with. If the parity-check fails, the Erase Tamper Detection Line is triggered.

16.1.1 Counter Unit

The Counter Unit (CU) generates counters C1 and C2 (used internally). In addition, the CU outputs Cn and flag CZ for use externally. The block diagram for the CU is shown in Figure 21:

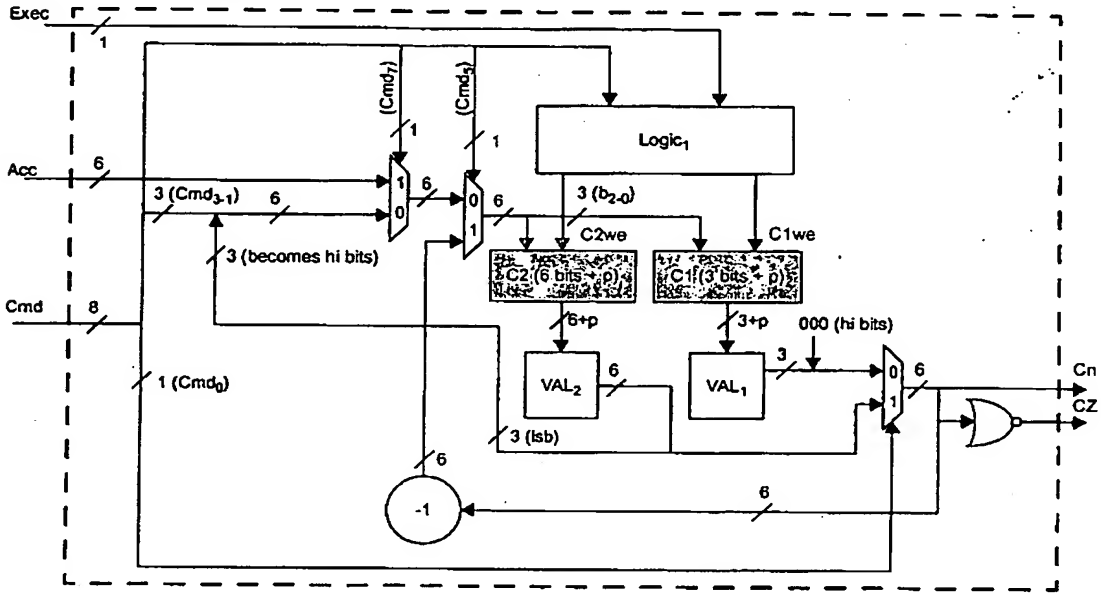


Figure 21. Block diagram for Counter Unit

Registers C1 and C2 are updated when they are the targets of a DBR, SC or ROR instruction. $Logic_1$ generates the control signals for the write enables as shown in the following pseudocode.

```
isDbrSc = (Cmd7-4 = DBR) ∨ (Cmd7-4 = SC)
isRorCn = (Cmd7-4 = ROR) ∧ (Cmd3-2 = 10)

CnWE = Exec ∧ (isDbrSc ∨ isRorCn)
C1we = CnWE ∧ ¬Cmd0
C2we = CnWE ∧ Cmd0
```

The single bit flag CZ is produced by the NOR of the appropriate C1 or C2 register for use during a DBR instruction. Thus CZ is 1 if the appropriate Cn value = 0.

The actual value written to C1 or C2 depends on whether the ROR, DBR or SC instruction is being executed. During a DBR instruction, the value of either C1 or C2 is decremented by 1 (with wrap). One multiplexor selects between the lower 6 bits of the Accumulator (for ROR instructions), and a 6-bit value for an SC instruction where the upper 3 bits = the low 3 bits from C2, and low 3 bits = low 3 bits from Cmd. *Note that only the lowest 3 bits of the operand are written to* C1.

The two VAL units are validation units connected to the Tamper Prevention and Detection circuitry (described in Section 10.3.5 on page 35), each with an OK bit. The OK bit is set to 1 on PORstL, and ORed with the ChipOK values from both Tamper Detection Lines each cycle. The OK bit is ANDed with each data bit that passes through the unit. All VAL units

also parity check the data to ensure the counters have not been tampered with. If a parity check fails, the Erase Tamper Detection Line is triggered.

In the case of VAL_1, the effective output from the counter C1 will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs.

In the case of VAL_2, the effective output from the counter C2 will always be 0 if the chip has been tampered with. This prevents an attacker from executing any looping constructs.

16.1.2 Calculate Next Address

This unit generates the address of the operand for the next instruction to be executed. It makes use of the Address Register Unit and PC to obtain base addresses, and the counters from the Counter Unit to assist in generating offsets from the base address.

This unit consists of some simple combinatorial logic, including an adder that adds a 6-bit number to a 10-bit number. The logic is shown in the following pseudocode.

```
isErase = (Cmd7-0 = ERA)
isSt = (Cmd7-4 = ST)
isAccRead = (Cmd7-6 = 10)

# First determine whether this is an immediate mode requiring PC+1
isJmpJsrDbrTbrImmed = (Cmd7-6 = 00) ∧ (¬Cmd5 ∨ (Cmd5-1 = 1x000))
isLia = (Cmd7-3 = LIA)
isLogImmed = ((Cmd7-6 = 11) ∧ ((Cmd5 ∨ Cmd4) ∧ (Cmd5-3 ≠ 111))) ∧ (Cmd1-0 = 10)
pcSel = Cycle ∨ (¬Cycle ∧ (isJmpJsrDbrTbrImmed ∨ isLogImmed ∨ isLia))

# Generate AnSel signal for the Address Register Unit
A0Sel = (isAccRead ∨ isSt) ∧ (¬Cmd3 ∨ (Cmd5-3 = 001))
AnSel1-0 = ¬A0Sel ∧ Cmd2-1

# The next address is either the new PC or must be generated
# (we require the base address from Address Register Unit)
nextRAMSel = AnDataOut8 ∧ ¬isErase
If (nextRAMSel)
   baseAdr = 00 | AnDataOut7-0# ram addresses are already word aligned
Else
   baseAdr = AnDataOut7-0 | 00# flash addresses are 4-byte aligned
EndIf
# Base address is now word (4-byte) aligned
# Now generate the offset amount to be added to the base address
selCn = (isAccRead ∨ isSt) ∧ (Cmd5 ∨ Cmd4) ∧ Cmd3

offset0 = (A0Sel ∧ Cmd0) ∨ (selCn ∧ Cn0)
offset1 = (A0Sel ∧ Cmd1) ∨ (selCn ∧ Cn1)
offset2 = (A0Sel ∧ Cmd2) ∨ (selCn ∧ Cn2)
offset5-3 = selCn ∧ Cn5-3
If (isErase)
   nextEffAdr11-4 = Acc7-0
   nextEffAdr3-0 = don't care
Else
   # now we can simply add the offset to the base address to get the effective adr
   nextEffAdr11-2 = baseAdr + offset # 10 bit plus 6 bit, with wrap = 10 bits out
   nextEffAdr1-0 = 0# word access, so lower bits of effadr are 0
EndIf
# Now generate the various signals for use during Cycle=1
# Note that these are only valid when pcSel is 0 (otherwise will read PC)
nextAccessMode0 = 1# want 32-bit access
nextAccessMode1 = nextRAMSel# ram or flash access (only valid if rd/wr/erase set)
nextAccessMode2 = isAccRead# pcSel takes care of LIA instruction
```

```
nextAccessMode3 = isSt# write access
nextAccessMode4 = isErase# erase page access
```

16.1.3 Address Register Unit

This unit contains 4 × 9-bit registers that are optionally cleared to 0 on PORstL. The 2-bit input AnSel selects which of the 4 registers to output on DataOut. When the writeEnable is set, the AnSel selects which of the 4 registers is written to with the 9-bit DataIn.

17 Program Mode Unit

The Program Mode Unit (PMU) is responsible for Program Mode and Trim Mode operations:

- Program Mode involves erasing the existing flash memory and loading the new program/data into the flash. The program that is loaded can be a bootstrap program if desired, and may contain additional program code to produce a digital signature of the final program to verify that the program was written correctly (e.g. by producing a CRC or, more likely, a SHA-1 signature of the entire flash memory.
- Trim Mode involves counting the number of internal cycles that have elapsed between the entry of Trim Mode (at the falling edge of the ack) and the receipt of the next byte (at the falling edge of the last bit before the ack) from the Master. When the byte is received, the current count value divided by 2 is transmitted to the Master.

The PMU relies on a fuse (implemented as the value of word 0 of the flash information block) to determine whether it is allowed to perform Program Mode operations. The purpose of this fuse is to prevent easy (or accidental) reprogramming of QA Chips once their purpose has been set. For example, an attacker may want to reuse chips from old consumables. If an attacker somehow bypasses the fuse check, the PMU will still erase all of flash before storing the desired program. Even if the attacker somehow disconnects the erasure logic, they will be unable to store a program in the flash due to the shadow nybbles.

The PMU contains an 8-bit buff register that is used to hold the byte being written to flash and a 12-bit adr register that is used to hold the byte address currently being written to.

The PMU is also used to load word 1 of the information block into a 32-bit register (combined from 8-bits of buff, 12-bits of adr, and a further 12-bit register) so it can be used to XOR all data to and from memory (both Flash and RAM) for future CPU accesses. This logic is activated only when the chip enters ActiveMode (so as not to access flash and possibly cause an erasure directly after manufacture since shadows will not be correct). The logic and 32-bit mask register is in the PMU to minimize chip area.

The PMU therefore has an asymmetric access to flash memory:

- writes are to main memory
- reads are from information block memory

The reads and writes are automatically directed appropriately in the MRU.

A block diagram of the PMU is shown in Figure 22.

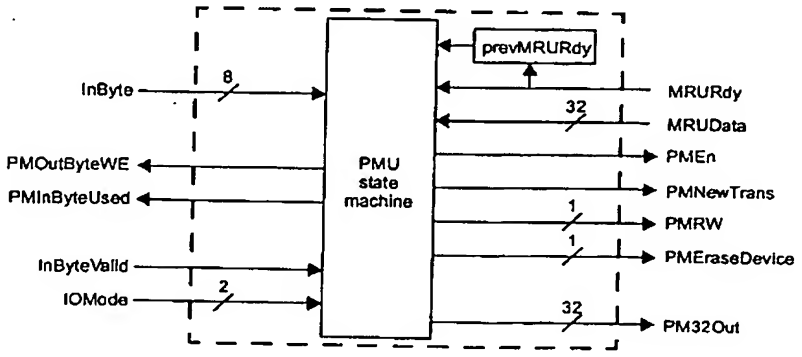


Figure 22. Block diagram of PMU

17.1 LOCAL STORAGE AND COUNTERS

The PMU keeps a 1-cycle delayed version of MRURdy, called prevMRURdy. It is used to generate PMNewTrans. Therefore each cycle the PMU performs the following task:

```
prevMRURdy ← MRURdy ∨ (state = loadByte)
```

The PMU also requires a 1-bit maskLoaded register. All registers are cleared to 0 on RstL.

17.2 STATE MACHINE

```
rstl
   prevMRURdy ← 0
   maskLoaded ← 0
   adr ← 0
   state ← idle
```

The **idle** state, entered after reset, simply waits for the IOMode to enter Program Mode, ActiveMode, or TrimMode. In state **wait4mode**, PMEn = 0. In state idle, PMEn = ¬maskLoaded. In all other states, PMEn = 1.

```
idle
   PMEn = ¬maskLoaded
   PMNewTrans = 0
   If ((IOMode = ActiveMode) ∧ MRURdy)
      If (maskLoaded)
         state ← wait4mode# no need to reload mask
      Else
         adr ← 4 # byte 4 is word 1 (the location of the XORMask)
         state ← getMask
      EndIf
   ElseIf ((IOMode = ProgramMode) ∧ MRURdy)# wait 4 access 2 fin
      maskLoaded ← 0# the mask is now invalid
      adr ← 0 # the location of the fuse is within 32-bit word 0
      state ← loadFuse
   ElseIf ((IOMode = TrimMode) ∧ MRURdy)# wait 4 access 2 finish
      maskLoaded ← 0# the mask is now invalid
      adr ← 0 # start the counter on entering TrimMode
      state ← trim
   Else
      state ← idle
   EndIf
```

The **wait4mode** state simply waits until for the current mode to finish and returns to idle.

```
wait4mode
   PMEn = 0
   PMNewTrans = 0
   If (IOMode = IdleMode)
      state ← idle
   Else
      state ← wait4mode
   EndIf
```

The **trim** state is where we count the number of cycles between the entry of the Trim Mode and the arrival of a byte from the Master. When the byte arrives from the Master, we send the resultant count:

```
trim
   # We saturate the adder at all 1s to make external trim control easier
   lastOne = adr0 ∧ adr1 ∧ ... adr11
   If (¬lastOne)
      adr = adr + 1# 12 bit incrementor
   EndIf
   # This logic simply causes the current adder value to be written to the
   # outByte when the inByte is received. The inByte is cleared when received
   # although it is not strictly necessary to do so
   PMOutByteWE = InByteValid# 0 in all other states
   PMInByteUsed = InByteValid# same as in loadByte state, 0 in all other states
   If (IOMode = IdleMode)
      state ← idle
   ElseIf (InByteValid)
      state ← wait4mode
   Else
      state ← trim
   EndIf
```

To check if we are allowed to program the device, load the 32-bit fuse value from word 0 of information memory in flash and compare it against the FuseSig constant (0x5555AAAA). If all is well we are allowed to enter the erase state (the first real step of programming).

```
loadFuse
   PMEn = 1
   PMNewTrans = prevMRURdy
   If (MRURdy)
      If (IOMode = ProgramMode)
         If (MRUData31-0 ≠ FuseSig)
            state ← erase
         Else
            state ← wait4mode
         EndIf
      Else
         state ← idle
      EndIf
   Else
      state ← loadFuse
   EndIf
```

The erase state erases the flash memory and then leads into the main programming states:

```
erase
   PMNewTrans = prevMRURdy
   PMEraseDevice = 1 # is 0 in all other states
   adr ← 0
   If (IOMode = IdleMode)
      state ← idle
   Else
      If (MRURdy)
         state ← loadByte
      Else
         state ← erase
      EndIf
   EndIf
```

Program Mode involves loading a series of 8-bit data values into the Flash. The PMU reads bytes via the IOU's InByte and InByteValid, setting MUInByteUsed as it loads data. The Master must send data slightly slower than the speed it takes to write to Flash to ensure that data is not lost.

```
loadByte        # Load in 1 bytes (1 word) from IO Unit
   PMNewTrans = 0
   PMInByteUsed = InByteValid# same as in TrimIn state, and 0 in all other states
   If (IOMode = IdleMode)
      state ← idle
   Else
      If (InByteValid)
         buff ← InByte
         state ← writeByte
      Else
         state ← loadByte
      EndIf
   EndIf

writeByte
   PMNewTrans = prevMRURdy
   PMRW = 0     # write. In all other states, PMRW = 1 (read)
   PM32Out7-0 = buff# data (can be tied to this)
   PM32Out19-8 = adr# can be tied to this
   PM32Out31-20 = 12bitReg# is always this (is don't care during a write)
   If (IOMode = IdleMode)
      state ← idle
   Else
      If (MRURdy)
         lastOne = adr0 ∧ adr1 ∧ ... adr11
         adr ← adr + 1# 12 bit incrementor
         If (lastOne)
            state ← wait4Mode
         Else
            state ← loadByte
         EndIf
      Else
         state ← writeByte
      EndIf
   EndIf
```

The **getMask** state loads up word 1 of the flash information block (bytes 4-7) into the 32-bit buffer so it can be used to XOR all data to and from memory (both Flash and RAM) for future CPU accesses.

```
getMask
   PMNewTrans = prevMRURdy
   PM32Out19-8 = adr# adr should = 4, i.e. word 1 which holds the CPU's mask
   PMRW = 1     # read (MUST be 1 in this state)
   If (IOMode = IdleMode)
      state ← idle
   Else
      If (MRURdy)
         buff ← MRUData7-0
         adr ← MRUData19-8
         12bitReg ← MRUData11-20
         maskLoaded ← 1
         state ← wait4mode
      Else
         state ← getMask
      EndIf
   EndIf
```

18 Memory Request Unit

The Memory Request Unit (MRU) provides arbitration between PMU memory requests and CPU-based memory requests.

The arbitration is straightforward: if the input PMEn is asserted, then PMU inputs are processed and CPU inputs are ignored. If PMEn is deasserted, the reverse is true.

A block diagram of the MRU is shown in Figure 23.

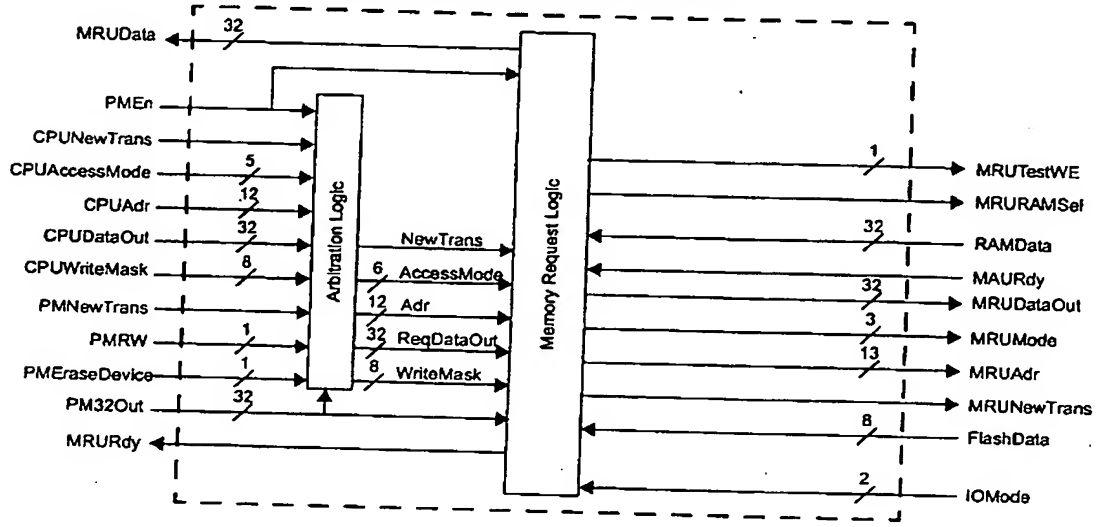


Figure 23. Block diagram of MRU

18.1 ARBITRATION LOGIC

The arbitration logic block provides arbitration between the accesses of the PM and the 8/32-bit accesses of the CPU via a simple multiplexing mechanism based on PMEn:

```
ReqDataOut31-8 = CPUDataOut31-8
If (PMEn)
   NewTrans = PMNewTrans
   AccessMode0 = PMRW# maps to 1 for reads (32 bits), 0 for writes (8 bits)
   AccessMode1 = 0# flash accesses only
   AccessMode2 = PMRW ∧ ¬PMEraseDevice# read has lower priority than erase
   AccessMode3 = ¬PMRW ∧ ¬PMEraseDevice# write has lower priority than erase
   AccessMode4 = 0# pageErase
   AccessMode5 = PMEraseDevice# erase everything (main & info block)
   WriteMask = 0xFF
   Adr = PM32Out19-8
   ReqDataOut7-0 = PM32Out7-0
Else
   NewTrans = CPUNewTrans ∧ (CPUAccessMode4-2 ≠ 000)
   AccessMode4-0 = CPUAccessMode
   AccessMode5 = 0 # cpu cannot ever erase entire chip
   WriteMask = CPUWriteMask
   Adr = CPUAdr
   ReqDataOut7-0 = CPUDataOut7-0
EndIf
```

18.2 MEMORY REQUEST LOGIC

The Memory Request Logic in the MRU implements the memory requests from the selected input. An individual request may involve outputting multiple sub-requests e.g. an 8-bit read consists of 2 × 4-bit reads (each flash byte contains a nybble plus its inverse).

The input accessMode bits are interpreted as follows:

Table 35. Interpretation of accessMode bits

bit #	description
0	0 = 8-bit access 1 = 32-bit access
1	0 = flash access 1 = RAM access this bit is only valid if bit 2, 3 or 4 is set
2	1 = read access
3	1 = write access
4	1 = erase page access
5	1 = erase entire (info and main) flash (only used within the MRU)

The MRU contains the following registers for general purpose flow control:

Table 36. Description of register settings

name	#bits	description
activeTrans	1	Is there a transaction still running? If so, then extraTrans and nextToXfer can be considered valid.
badUntilRestart	1	0 = memory (flash and ram) reads work correctly 1 = memory (flash and ram) reads return 0 Gets set whenever illChip gets set, and remains set until a soft restart occurs i.e. IOMode transitioned through idle.
extraTrans	1	Determines whether there is an additional sub-transaction to perform. e.g. a 32 bit read from flash involves 4 sub-transactions in the case of 8-bit accesses, and 8 sub-transactions in the case of 4-bit accesses.
illChip	1	0 = 15 consecutive bad reads have not occurred 1 = 15 consecutive bad reads have occurred
nextToXfer	3	The next element (byte or nybble) number to transfer to/from memory
restartPending	1	Did IOMode change while the transaction was being processed?
retryCount	4	Number of times that a byte has been read badly from flash. When a byte has been read badly 15 consecutive times *illChip* will be set.
retryStarted	1	0 = no retries encountered yet for this read 1 = retries have been encountered - retryCount holds the number of retries The retryStarted register is used to stop retryCount being cleared on good reads - thus keeping a record of the last number of retries on a bad read.

Table 37 lists the registers specifically for testing flash. Although the complete set of flash test registers is in both the MRU and MAU (group 0 is in the MRU, groups 1 and 2 are in the MAU), all the decoding takes place from the MRU.

Table 37. Flash test registers settable from CPU when the RAM address is > 128[a]

adr bits	bits	name	description
0	0	shadowsOff	0 = regular shadowing (nybble based access to flash) 1 = shadowing disabled, 8-bit direct accesses to flash.
	1	hiFlashAdr	Only valid when shadowsOff = 1 0 = accesses are to lower 4Kbytes of flash 1 = accesses are to upper 4 Kbytes of flash
	2	infoBlockSel	0 = 32-bit accesses to lower 128[b] / 64[c] bytes of flash is to main memory 1 = accesses are to information block
1	3	enableFlashTest	0 = keep flash test register within the TSMC flash IP in its reset state 1 = enable flash test register to take on non-reset values.
	8-4	flashTest	Internal 5-bit flash test register within the TSMC flash IP (SFC008_08B9_HE). If this is written with 0x1E, then subsequent writes will be according to the TSMC write test mode. You must write a non-0x1E value or reset the register to exit this mode.
2	28-9	flashTime	When timerSel is 1, this value is used for the duration of the program cycle within a standard flash write or erasure. 1 unit = 16 clock cycles (16 × 100ns typical). Regardless of timerSel, this value is also used for the time-out following power down detection before the QA Chip resets itself. 1 unit = 1 clock cycle (= 100ns typical). *Note that this means the programmer should set this to an appropriate value (e.g. 5 µs), just as the localId needs to be set.*
	29	timerSel	0 = use internal (default) timings for flash writes & erasures 1 = use flashTime for flash writes and erasures

a. This is from the programmer's perspective. Addresses sent from the CPU are byte aligned, so the MRU needs to test bit n+2. Similarly, checking DRAM address > 128 means testing bit 7 of the address in the CPU, and bit 9 in the MRU.
b. unshadowed
c. shadowed

18.2.1 Reset

Initialization on reset involves clearing all the flags:

```
MRURdy = 0# can't process anything at this point
activeTrans ← 0
extraTrans ← 0
illChip ← 0
badUntilRestart ← 0
restartPending ← 0
retryCount ← 0
retryStarted ← 0
nextToXfer ← 0# don't care
shadowsOff ← 0
hiFlashAdr ← 0
infoBlockSel ← 0# used to generate MRUMode2
```

18.2.2 Main logic

The main logic consists of waiting for a new transaction, and starting an appropriate sub-transaction accordingly, as shown in the following pseudocode:

```
# Generate some basic signals for use in determining accessPatterns
Is32Bit = AccessMode0
Is8Bit = ¬AccessMode0
IsFlash = ¬AccessMode1
IsRAM = AccessMode1
IsRead = AccessMode2
IsWrite = AccessMode3
noShadows = shadowsOff
doShadows = IsFlash ∧ ¬noShadows
powerGood = (IOMode ≠ IdleMode)
startOfSubTrans = (NewTrans ∨ extraTrans) ∧ powerGood
doingTrans = startOfSubTrans ∨ activeTrans
IsInvalidRAM = doingTrans ∧ IsRAM ∧ (Adr9 ∨ (Adr8 ∧ Adr7))
IsTestModeWE = doingTrans ∧ IsRAM ∧ IsWrite ∧ Adr9
IsTestReg0 = IsTestModeWE ∧ Adr3#write to flash test register - bit 1 of word adr
IsTestReg1 = IsTestModeWE ∧ Adr4#write to flash test register - bit 2 of word adr
MRUTestWE = IsTestReg0 ∨ IsTestReg1
IsPageErase = AccessMode4
IsDeviceErase = AccessMode5 ∨ (IsTestModeWE ∧ (Adr8-2 = 0001000)) # bit 9 not req
IsErase = IsDeviceErase ∨ IsPageErase
MRURAMSel = IsRAM ∧ ¬MRUTestWE ∧ ¬IsDeviceErase
IsInfBlock = (PMEn ∧ (IsDeviceErase ∨ IsRead)) ∨
             (¬PMEn ∧ infoBlockSel ∧
                (IsDeviceErase ∨ (IsFlash ∧ (Adr11-7 = 0) ∧ ¬(Adr6 ∧ doShadows))))

# Which element (byte or nybble) are we up to xferring?
If (NewTrans)
   toXfer = 0
Else
   toXfer = nextToXfer
EndIf

# Form the address that goes to the outside world
If (IsFlash ∧ noShadows)
   byteCount = toXfer1-0
   MRUAdr12 = hiFlashAdr# upper or lower block of 4Kbytes of flash
   MRUAdr11-2 = Adr11-2# word #
   MRUAdr1-0 = (Adr1-0 ∧ (¬Is32Bit|¬Is32Bit)) ∨ byteCount# byte
Else
   byteCount = toXfer2-1
   MRUAdr12-3 = Adr11-2# word #
   MRUAdr2-1 = (Adr1-0 ∧ (¬Is32Bit|¬Is32Bit)) ∨ byteCount# byte
   MRUAdr0 = toXfer0#nybble
EndIf

# Assuming a write, are we allowed to write to this address?
writeEn = SelectBit[WriteMask, ((MRUAdr2 ∧ doShadows)| MRUAdr1-0)]# mux: 1 from 8

# Generate the 4-bit mask to be used for XORing during CPU access to flash
baseMask = SelectNybble[PM32Out, MRUAdr2-0]# mux selects 4 bits of 32
If (PMEn)
   theMask = 0
Else
   theMask = baseMask# we only use mask for CPU accesses to flash
EndIf

# Select a byte (and nybble) from the data for writes
baseByte = SelectByte[ReqDataOut, byteCount]# mux: 8 bits from 32
baseNybble = SelectNybble[baseByte, toXfer0]# mux: 4 bits from 8
```

```
outNybble = baseNybble ⊕ theMask# only used when nybble writing

# Generate the data on the output lines (doesn't matter for reads or erasures)
MRUDataOut31-8 = ReqDataOut31-8# effectively don't care for flash writes
If (doShadows)
   MRUDataOut7 = ¬outNybble3
   MRUDataOut6 = outNybble3
   MRUDataOut5 = ¬outNybble2
   MRUDataOut4 = outNybble2
   MRUDataOut3 = ¬outNybble1
   MRUDataOut2 = outNybble1
   MRUDataOut1 = ¬outNybble0
   MRUDataOut0 = outNybble0
Else
   MRUDataOut7-0 = baseByte
EndIf

# Setup MRUMode
allowTrans = IsRAM ∨ IsRead ∨ (IsWrite ∧ writeEn) ∨ IsErase
If (doingTrans)
   MRUMode2 = IsInfBlock
   MRUMode1 = IsErase ∨ IsTestReg1
   MRUMode0 = IsDeviceErase ∨ (¬IsWrite ∧ ¬IsPageErase) ∨ IsTestReg0
   MRUNewTrans = startOfSubTrans ∧ allowTrans
Else
   MRUMode2-0 = 001 # read (safe)
   MRUNewTrans = 0
EndIf

# Generate the effective nybble read from flash (this may not be used).
# When there is a shadowFault (i.e. if non-erased memory and shadows are invalid)
# we consider it a bad read when an 8-bit read, or when writeMask0 is 0.
# Note: we always substitute the upper nybble of WriteMask for the non-valid data,
# but only flag a read error if WriteMask0 is also 1. When the data is erased, we
# return 0 regardless of WriteMask0.
finishedTrans = doingTrans ∧ MAURdy
finishedFlashSubTrans = finishedTrans ∧ IsFlash ∧ ¬IsErase
isWrittenFlash = (FlashData7-0 ≠ 11111111)# flash is erased to all 1s
If (isWrittenFlash ∧ ((FlashData7,5,3,1 ⊕ FlashData6,4,2,0) ≠ 1111))
   inNybble3-0 = WriteMask7-4
   badRead = finishedFlashSubTrans ∧ IsRead ∧ (Is8Bit ∨ ¬WriteMask0) ∧ doShadows
Else
   inNybble3,2,1,0 = (theMask3,2,1,0 ⊕ FlashData6,4,2,0) ∧ isWrittenFlash
   badRead = 0
EndIf

# Present the resultant data to the outside world
If (IsErase ∨ IsWrite ∨ ¬doingTrans ∨ IsInvalidRAM ∨ badRead ∨ badUntilRestart)
   MRUData0 = IsInvalidRAM ∧ illChip
   MRUData4-1 = retryCount ∧ (IsInvalidRAM ∧ Adr2)# mask all 4 count bits
   MRUData31-5 = 0# also ensures a read that is bad returns 0
ElseIf (IsRAM)
   MRUData31-24 = SelectByte[RAMData, (Adr1-0 ∨ Is32Bit|Is32Bit)]# mux: 8 from 32
   MRUData23-0 = RAMData23-0# lsbs remain unchanged from RAM
ElseIf (doShadows)
   MRUData31-28 = inNybble
   MRUData27-0 = buff27-0
Else
   MRUData31-24 = FlashData
   MRUData23-0 = buff27-4
EndIf

# Shift in the data for the good reads - either 4 or 8 bits (writes = don't care)
If (finishedFlashSubTrans ∧ ¬badRead)
   buff3-0 ← buff7-4# shift right 4 bits
```

```
      If (doShadows)
         buff23-4 ← buff27-8    # shift right 4 bits
         buff27-24 ← inNybble
      Else
         buff19-4 ← buff27-12# shift right 8 bits, buff3-0 is don't care
         buff27-20 ← FlashData
      EndIf
   EndIf

   # Determine whether or not we need a new sub-transaction. We only need one if:
   # * we're doing 8 bit reads that are shadowed
   # * we're doing 32 bit reads and we've done less than 4 or 8 (sh vs non-sh)
   # * we got a bad read from flash and we need to retry the read (jic was a glitch)
   moreAdrsToGo = (¬toXfer0 ∧ ((Is8Bit ∧ doShadows) ∨ Is32Bit)) ∨
                  (¬toXfer1 ∧ Is32Bit) ∨ (¬toXfer2 ∧ Is32Bit ∧ doShadows)
   needToRetryRead = badRead ∧ (¬retryStarted ∨ (retryCount ≠ 1111))
   extraTrans_in = finishedFlashSubTrans ∧ (moreAdrsToGo ∨ needToRetryRead)
   nextToXfer ← toXfer + (finishedFlashSubTrans ∧ (IsWrite ∨ ¬needToRetryRead))

   # generate our rdy signal and state values for next cycle
   MRURdy = ¬doingTrans ∨ (doingTrans ∧ MAURdy ∧ ¬extraTrans_in)
   extraTrans ← extraTrans_in
   activeTrans ← ¬MRURdy# all complete only when MRURdy is set

   # Capture writes to the local registers
   # Ensure that we won't have problems restarting a program!
   If (MRURdy ∧ (restartPending ∨ ¬powerGood))# note MRURdy (end of word write!)
      shadowsOff ← 0
      hiFlashAdr ← 0
      infoBlockSel ← 0
      restartPending ← 0
      badUntilRestart ← 0
      retryStarted ← 0# clear flag so will be ok for the next read
   Else
      If (doingTrans ∧ ¬powerGood)
         restartPending ← 1# record for later use
      EndIf
      If (IsTestModeWE ∧ Adr2)# the other writes are taken care of by the MAU
         shadowsOff ← ReqDataOut0
         hiFlashAdr ← ReqDataOut1
         infoBlockSel ← ReqDataOut2
      EndIf
      If (MAURdy)
         If (IsTestModeWE ∧ (Adr5-2 = 0000))
            illChip ← 0
            retryCount ← 0
         Else
            badUntilRestart_in = badUntilRestart ∨
                     (badRead ∧ retryStarted ∧ (retryCount = 1111))
            illChip ← illChip ∨ badUntilRestart_in
            badUntilRestart ← badUntilRestart_in
            If (badRead)
               retryCount ← (retryCount ∧ retryStarted) + 1# AND all 4 bits
               retryStarted ← 1
            Else
               retryStarted ← 0# clear flag so will be ok for the next read
            EndIf
         EndIf
      EndIf
   EndIf
```

19 Memory Access Unit

The Memory Access Unit (MAU) takes memory access control signals and turns them into RAM accesses and flash access strobed signals with appropriate duration.

A new transaction is given by MRUNewTrans. The address to be read from or written to is on MRUAdr, which is a nybble-based address. The MRUAdr (13-bits) is used as-is for Flash addressing. When MRURAMSel is true, then the RAM address (RAMAdr) is taken from bits 9-3 of MRUAdr. The data to be written is on MRUData.

The return value MAURdy is set when the MAU is capable of receiving a new transaction the following cycle. Thus MAURdy will be 1 during the final cycle of a flash or ram access, and should be 1 when the MAU is idle. MAURdy should only be 0 during startup or when a transaction has yet to finish.

When MRURAMSel = 1, the access is to RAM, and MRUMode has the following interpretation:

Table 38. Interpretation of MRUMode[a] for RAM accesses

MRUMode bits	Action
xx0	doWrite
xx1	doRead

a. $MRUMode_{2-1}$ is ignored for RAM accesses

When MRURAMSel = 0, the access is to flash. If MRUTestWE = 0, then the access is to regular flash memory, as given by MRUMode:

Table 39. Interpretation of MRUMode for regular flash accesses[a]

MRUMode bits 1-0	Action when MRUMode$_2$ = 0	Action when MRUMode$_2$ = 1
00	doWrite (main memory)	doWrite (info block)
01	doRead (main memory)	doRead (info block)
10	doErasePage (main memory)	doErasePage (info block)
11	doEraseDevice (main memory)	doEraseDevice (*both* blocks)

a. $MRUMode_2$ can be directly interpreted by the MAU as the IFREN signal required for embedded flash block SFC008_08B9_HE

If MRUTestWE is 1, then $MRUMode_2$ will also be 0, and the access is to a flash test register, as given by MRUMode:

Table 40. Interpretation of MRUMode for flash test register write accesses

MRUMode bits[a]	Action
xx1	If ($MRUData_3$ = 0), tie the flash IP test register to its reset state If ($MRUData_3$ = 1), take the flash IP test register out of reset state, and write $MRUData_{8-4}$ to the 5-bit flash test register within the flash IP (SFC008_08B9_HE)
x1x	Write $MRUData_{28-9}$ to the internal 20-bit alternate-counter-source register flashTime, and $MRUData_{29}$ to the corresponding 1-bit test register timerSel.

a. $MRUMode_2$ will always be 0 when MRUTestWE = 1.

19.1 IMPLEMENTATION

The MAU consists of logic that calculates MAURdy, and additional logic that produces the various strobed signals according to the TSMC Flash memory SFC0008_08B9_HE; refer to this datasheet [4] for detailed timing diagrams. Both main memory and information blocks can be accessed in the Flash. The Flash test modes are also supported as described in [5] and general application information is given in [6].

The MAU can be considered to be a RAM control block and a flash control block, with appropriate action selected by MRURAMSel. For all modes except read, the Flash requires wait states (which are implemented with a single counter) during which it is possible to access the RAM. Only 1 transaction may be pending while waiting for the wait states to expire. Multiple bytes may be written to Flash without exiting the write mode.

The MAU ensures that only valid control sequences meeting the timing requirements of the Flash memory are provided. A write time-out is included which ensures the Flash cannot be left in write mode indefinitely; this is used when the Flash is programmed via the IO Unit to ensure the X address does not change while in write mode. Otherwise, other units should ensure that when writing bytes to Flash, the X address does not change. The X address is held constant by the MAU during write and page erase modes to protect the Flash. If an X address change is detected by the MAU during a Flash write sequence, it will exit write mode allowing the X address to change and re-enter write mode. Thus, the data will still be written to Flash but it will take longer.

When the Flash is not being used, MAU sets the control signals to put it into standby which minimises power consumption.

The MAU assumes no new transactions can start while one is in progress and all inputs must remain constant until MAU is ready.

19.2 FLASH TEST MODE

MAU also enables the Flash test mode register to be programmed which allows various production tests to be carried out. If MRUTestWE = 1, transactions are directed towards the test mode register. Most of the tests use the same control sequences that are used for normal operation except that one time value needs to be changed. This is provided by the flashTime register that can be written to by the CPU allowing the timer to be set to a range of values up to more than 1 second. A special control sequence is generated when the test mode register is set to 0x1E and is initiated by writing to the Flash.

Note that on reset, timeSel and flashTime are both cleared to 0. The 5-bit flash test register within the TSMC flash IP is also reset by setting TMR =1. When MRUTestWE = 1, any open write sequence is closed even if the write is not to the 5-bit flash test register within the TSMC flash IP.

19.3 FLASH POWER FAILURE PROTECTION

Power could fail at any time; the most serious consequence would be if this occurred during writing to the Flash and data became corrupted in another location to that being written to. The MAU will protect the Flash by switching off the charge pump (high voltage supply used for programming and erasing) as soon as the power starts to fail. After a time delay of about 5µs (programmable), to allow the discharge of the charge pump, the QA chip will be reset whether or not the power supply recovers.

19.4 FLASH ACCESS STATE MACHINE

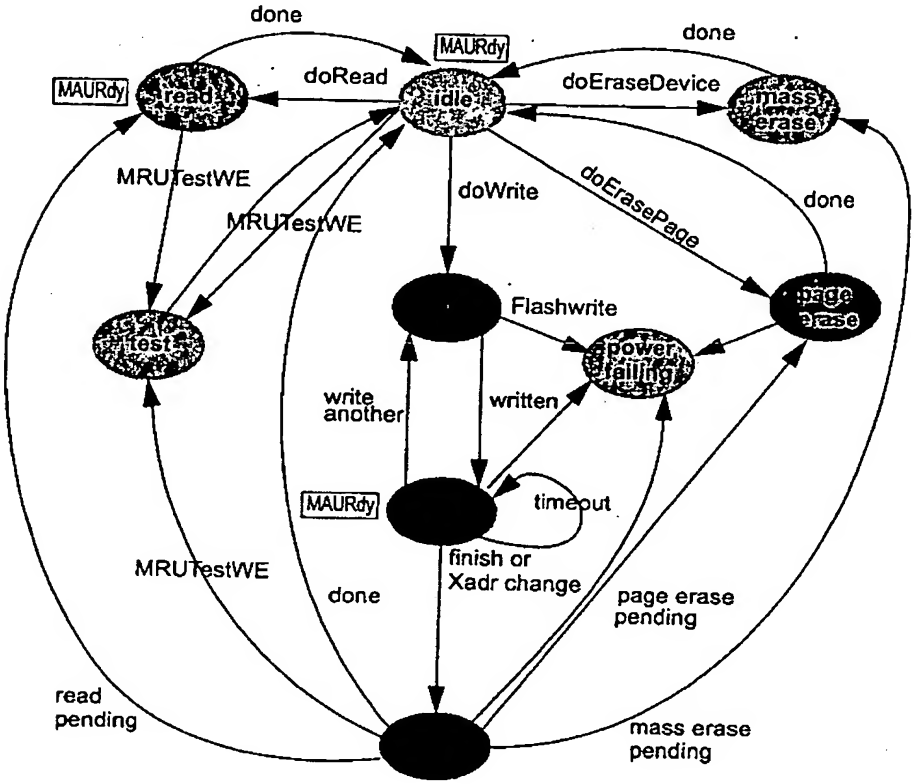


Figure 24. Simplified MAU state machine

19.5 INTERFACE

Table 41. MAU Interface description

Signal name	I/O	Description
Clk	In	System clock.
RstL	In	System reset (active low).
MRUNewTrans	In	Flag indicating MRU wishes to start a new transaction. May only be asserted (= 1) when MAURdy = 1. All inputs below must be held constant until MAU is ready.
MRURAMSel	In	1 = RAM, 0 = Flash.
$MRUMode_{2-0}$	In	Type of transaction to be performed.
$MRUAdr_{12-0}$	In	Memory address from the MRU.
$MRUDataOut_{31-0}$	In	Data used to control and set test modes and timing.
MRUTestWE	In	Flag indicating test mode transactions.
PwrFailing	In	Flag indicating possible power failure in progress.
MAURdy	Out	The MAU is ready when MAURdy = 1. It is always set for RAM transactions and held low during Flash wait states.
RAMWE	Out	RAM write when RAMWE = 0 (Artisan Synchronous SRAM).
MemClk	Out	Inverted system clock to the RAM (required to meet timing).
$FlashCtrl_{8-0}$	Out	Control signals to the Flash. IFREN = information block enable, not used always = 0 XE = X address enable YE = Y address enable SE = sense amplifier enable (read only) OE = output enable (read only), hi-Z when OE = 0 PROG = program (write bytes) NVSTR = enables all write and erase modes ERASE = page erase mode MAS1 = mass erase mode
TMR	Out	TMR = Register reset for test mode
$RAMAdr_{6-0}$	Out	RAM address in the range 0 to 95.
$FlashAdr_{12-0}$	Out	Flash address, full range.
MAURstOutL	Out	Activates the global reset, RstL.

19.6 CALCULATION OF TIMER VALUES

Set and calculate timer initialisation values based on Flash data sheet values, clock period and clock range.

```
# Note: Flash data sheet gives minimum timings
# Delays greater than 1 clock cycle

   clock_per     = 100     # ns

  Flash_Tnvs    = 7500    # ns
  Flash_Tnvh    = 7500    # ns
  Flash_Tnvh1   = 150     # us
  Flash_Tpgs    = 10      # us
  Flash_Tpgh    = 100     # ns
  Flash_Tprog   = 30      # us
  Flash_Tads    = 100     # ns
```

```
  Flash_Tadh    = 30      # us  # Byte write timeout
  Flash_Trcv    = 1500    # ns
  Flash_Thv     = 6       # ms  # Not currently used
  Flash_Terase  = 30      # ms
  Flash_Tme     = 300     # ms

# Derive maximum counts (-1 since state machine is synchronous)
  FLASH_NVS    = Flash_Tnvs/clock_per - 1
  FLASH_NVH    = Flash_Tnvh/clock_per - 1
  FLASH_NVH1   = Flash_Tnvh1*1000/clock_per - 1
  FLASH_PGS    = Flash_Tpgs*1000/clock_per - 1
  FLASH_PGH    = Flash_Tpgh/clock_per - 1
  FLASH_PROG   = Flash_Tprog*1000/clock_per - 1
  FLASH_ADS    = Flash_Tads/clock_per - 1
  FLASH_ADH    = Flash_Tadh*1000/clock_per - 1
  FLASH_RCV    = Flash_Trcv/clock_per - 1
  FLASH_HV     = Flash_Thv*1000000/clock_per - 1
  FLASH_ERASE  = Flash_Terase*1000000/clock_per - 1
  FLASH_ME     = Flash_Tme*1000000/clock_per - 1

  count_size = 24 # Number of bits in timer counter (newCount) determined by Tme
```

19.7 DEFAULTS

Defaults to use when no action is specified.

```
   FlashTransPendingSet = 0
   FlashTransPendingReset = 0
  TMRSet = 0
  TMRRst = 0
  STLESet = 0
  STLERst = 0
  TestTimeEn = 0
  IFREN = FlashXadr7
   XE = 0
   YE = 0
   SE = 0
   OE = 0
   PROG = 0
   NVSTR = 0
   ERASE = 0
   MAS1 = 0
  MAURstOutL = 1

   If (accessCount ≠ 0)
     newCount =accessCount - 1 # decrement unless instructed otherwise
   Else
     newCount =0
   EndIf
```

19.8 RESET

Initialise state and counter registers.

```
# asynchronous reset (active low)
      state ← idle
      accessCount ← 1
      countZ ← 0
      XadrReg ← 0
      FlashTransPending ← 0
      TestTime ← 0
      TMR ← 1
      STLEFlag ← 0
```

19.9 STATE MACHINE

The state machine generates sequences of timed waveforms to control the operation of the Flash memory.

```
      idle
        newCount = 0
        If (FlashNewTrans) # Flash starting conditions
         If (¬MRUTestWE)
            Switch ( MRUModeint)
            Case doWrite:
               state ← writeNVS
                newCount = FLASH_NVS
            Case doRead:
                YE = 1
                SE = 1
                OE = 1
                XE = 1
                state = readByte
                newCount = 0
            Case doErasePage:
                state ← pageErase
                newCount = FLASH_NVS
            Case doEraseDevice:
               state ← massErase
                newCount = FLASH_NVS
            EndSwitch
         Else
            state ←TM0
         EndIf
        EndIf

####################
# Flash byte read #
####################

      readByte
        YE = 1
        SE = 1
        OE = 1
        XE = 1
        If (¬FlashNewTrans)
          state ← idle # exit read mode
          newCount = 0 # default is to stay in current state
         Else #what next? Must go immediately as MAU is ready
         If (¬MRUTestWE)
            Switch ( MRUModeint)
```

```
                Case doWrite:
                   state ← writeNVS
                   newCount = FLASH_NVS
                Case doRead: # just stay here in readByte
                Case doErasePage:
                   state ← pageErase
                   newCount = FLASH_NVS
                Case doEraseDevice:
                   state ← massErase
                   newCount = FLASH_NVS
                EndSwitch
             Else
                state ←TM0
             EndIf
            EndIf

##############################
# Flash page erase sequence #
##############################

      pageErase
        ERASE = 1
        XE = 1
         If (¬PwrFailing)
        If (countZ)
          newCount = FLASH_ERASE
          state ← pageEraseERASE
        EndIf
       Else
          newCount = TestTime19-0
          state ← Help1
       EndIf
       pageEraseERASE
        ERASE = 1
        NVSTR = 1
        XE = 1
         If (¬PwrFailing)
        If (countZ)
          newCount = FLASH_NVH
          state ← pageEraseNVH
        EndIf
       Else
          newCount = TestTime19-0
          state ← Help1
       EndIf
       pageEraseNVH
        NVSTR = 1
        XE = 1
         If (¬PwrFailing)
        If (countZ)
          newCount = FLASH_RCV
          state ← RCVPM
        EndIf
       Else
          newCount = TestTime19-0
          state ← Help1
       EndIf
       RCVPM
        If (countZ)
          newCount = 0
          state ← idle # exit
        EndIf

##############################
```

```
# Flash mass erase sequence #
##############################

      massErase
        MAS1 = 1
        ERASE = 1
      XE = 1
        If (countZ)
         If (¬TestTime20)
            newCount = FLASH_ME
         Else
            newCount = TestTime19-0 | 0000
         EndIf
         state ← massEraseME
        EndIf
      massEraseME
        MAS1 = 1
        ERASE = 1
        NVSTR = 1
      XE = 1
        If (countZ)
          newCount = FLASH_NVH1
          state ← massEraseNVH1
        EndIf
      massEraseNVH1
        MAS1 = 1
        NVSTR = 1
      XE = 1
        If (countZ)
          newCount = FLASH_RCV
          state ← RCVPM
        EndIf

##############################
# Flash byte write sequence #
##############################

      writeNVS
        PROG = 1
        XE = 1
         If (¬PwrFailing)
         If (countZ)
            If (¬STLEFlag)
               newCount = FLASH_PGS
               state ← writePGS
            Else
               newCount = TestTime19-0 | 0000
               state ← STLE0
            EndIf
        EndIf
      Else
         newCount = TestTime19-0
         state ← Help1
      EndIf
      writePGS
        PROG = 1
        NVSTR = 1
        XE = 1
      If (¬PwrFailing)
         If (countZ)
            newCount = FLASH_ADS
            state ← writeADS
         EndIf
      Else
```

```
    newCount = TestTime19-0
    state ← Help1
 EndIf
 writeADS # Add Tads to Tpgs
   PROG = 1
   NVSTR = 1
   XE = 1
  FlashTransPendingReset = 1
 If (¬PwrFailing)
   If (countZ)
    If (¬TestTime20)
       newCount = FLASH_PROG
    Else
       newCount = TestTime19-0 | 0000
    EndIf
       state ← writePROG
   EndIf
 Else
    newCount = TestTime19-0
    state ← Help1
 EndIf
 writePROG
   PROG = 1
   NVSTR = 1
   YE = 1
   XE = 1
 If (¬PwrFailing)
   If (countZ)
     newCount = FLASH_ADH
     state ← writeADH
   EndIf
 Else
    newCount = TestTime19-0
    state ← Help2
 EndIf
 writeADH
   PROG = 1
   NVSTR = 1
   XE = 1
  FlashTransPendingSet = PlashTransPending ∨ FlashNewTrans
 If (¬PwrFailing)
   If (¬FlashNewTrans)
     If (countZ)-- Gracefull exit after timeout
       newCount = FLASH_PGH
       state ← writePGH
     EndIf
   Else
     -- Do something as there is a new transaction
     If (( MRUModeint = doWrite) ∧ (¬XadrCh))
       newCount = FLASH_ADS -- Write another byte
       state ← writeADS
     Else
       newCount = FLASH_PGH -- Exit as new trans is not Flash write
       state ← writePGH
     EndIf
   EndIf
 Else
    newCount = TestTime19-0
    state ← Help1
 EndIf
 writePGH
   PROG = 1
   NVSTR = 1
   XE = 1
```

```
 FlashTransPendingSet = FlashTransPending v FlashNewTrans
If (¬PwrFailing)
  If (countZ)
    newCount = FLASH_NVH
    state ← writeNVH
  EndIf
Else
   newCount = TestTime19-0
   state ← Help1
EndIf
writeNVH
  NVSTR = 1
  XE = 1
 FlashTransPendingSet = FlashTransPending v FlashNewTrans
If (¬PwrFailing)
  If (countZ)
    newCount = FLASH_RCV
    state ← RCV
  EndIf
Else
   newCount = TestTime19-0
   state ← Help1
EndIf
RCV
 FlashTransPendingSet = FlashTransPending v FlashNewTrans
  If (countZ)
    newCount = 0
   FlashTransPendingReset = 1
    If (FlashTransPending)
    If (¬MRUTestWE)
      Switch ( MRUModeint) # MRUMode changed?, Flash transaction pending
        # This only happens if the RAM has been accessed following
        # Flash write
      Case doWrite:
            state ← writeNVS # Start writing again
            newCount = FLASH_NVS
      Case doRead:
          YE = 1
          SE = 1
          OE = 1
          XE = 1
          state ← readByte
          newCount = 0
      Case doErasePage:
          state ← pageErase
          newCount = FLASH_NVS
      Case doEraseDevice:
          state ← massErase
          newCount = FLASH_NVS
      EndSwitch
    Else
      state ←TM0
    EndIf
    Else
      state ← idle
    EndIf
  EndIf

#######################
# Test mode sequence #
#######################

     TM0 # Needed this due to delay on TMR
       IFREN = 0
```

```
    state ← idle # default
    If ( MRUModeint1)
      TestTimeEn  = 1
    EndIf
    If ( MRUModeint0)
      If (¬MRUDataOut3)
        TMRSet = 1
        STLERst = 1 # Reset flag as leaving test mode
      Else
        If (MRUDataOut8-4 = 11110)
          STLESet = 1
        Else
          STLERst = 1
        EndIf
        TMRRst = 1
        state ← TM1 # Will get priority
      EndIf
    EndIf
  TM1
    IFREN = 0
    state ← TM2
  TM2
    NVSTR = 1
    SE    = 1
    IFREN = 0
    state ← TM3
  TM3
    NVSTR = 1
    SE    = 1
    MAS1  = MRUDataOut4
    IFREN = MRUDataOut5
    XE    = MRUDataOut6
    YE    = MRUDataOut7
    ERASE = MRUDataOut8
    TMRSet = 1
    state ← TM4
  TM4
    NVSTR = 1
    SE    = 1
    MAS1  = MRUDataOut4
    IFREN = MRUDataOut5
    XE    = MRUDataOut6
    YE    = MRUDataOut7
    ERASE = MRUDataOut8
    TMRRst = 1
    state ← TM5
  TM5
    NVSTR = 1
    SE    = 1
    MAS1  = MRUDataOut4
    IFREN = MRUDataOut5
    XE    = MRUDataOut6
    YE    = MRUDataOut7
    ERASE = MRUDataOut8
    state ← TM6
  TM6
    NVSTR = 1
    SE    = 1
    state ← idle

############################################################
# Reverse tunneling and thin oxide leak test sequence #
```

```
##########################################################

      STLE0
        XE = 1
        PROG = 1
        NVSTR = 1
        If (countZ)
          newCount = FLASH_NVH
          state ← STLE1
        EndIf
      STLE1
        XE = 1
        NVSTR = 1
        If (countZ)
          newCount = FLASH_RCV
          state ← STLE2
        EndIf
      STLE2
        If (countZ = 1)
          newCount = 0
          state ← idle
        EndIf

############################
# Emergency instructions #
############################

      Help1 # MAURdy -> 0 to hold MAU inputs constant, if not too late
        XE = 1
        If (countZ)
          newCount = 0
          state ← Goodbye
        EndIf
      Help2 # MAURdy -> 0 to hold MAU inputs constant, if not too late
        XE = 1
        YE = 1
        If (countZ)
          newCount = 0
          state ← Goodbye
        EndIf
      Goodbye
        XE = 1 # Prevents Flash timing violation
        MAURstOutL = 0 # Reset whole chip whether power fails
         # nothing else to do                or recovers
```

19.10 CONCURRENT LOGIC

```
accessCount ← newCount # update accessCount every cycle
XadrReg ← FlashXAdr # store the previous X address

If (FlashTransPendingReset)
   FlashTransPending ← 0 # Reset flag (has priority)
Else
   If (FlashTransPendingSet)
      FlashTransPending ← 1 # Set flag
   EndIf
EndIf
If (TestTimeEn)
     TestTime ← MRUDataOut29-9
EndIf
If (TMRSet) -- SRFF for TMR
```

```
      TMR ← 1
   Else
      If (TMRRst)
          TMR ← 0
        EndIf
   EndIf
   If (STLERst) -- SRFF for STLE tests
      STLEFlag ← 0
   Else
      If (STLESet)
         STLEFlag ← 1
      EndIf
   EndIf

  countZ ← 1 When (newCount = 0) Else 0
  FlashRdy = 1 When (
    ((state = idle) ∧ ((¬FlashNewTrans) ∨( MRUModeint = doRead)))
   ∨((state = readByte) ∧( MRUModeint = doRead)) # another read
   ∨(((state = writeADH)
   ∨ (state = writePGH)
   ∨ (state = writeNVH)
   ∨ (state = RCV)) ∧ (¬FlashTransPendingSet))
    ) Else 0

   FlashNewTrans = MRUNewTrans ∧(¬MRURAMSel)
   RAMNewTrans = MRUNewTran ∧MRURAMSel
   IsValidAccess = ¬(¬RAMNewTrans ∨ MRUAdr10 ∨(MRUAdr9 ∧MRUAdr8))
  MAURdy = FlashRdy When (¬MRURAMSel) Else 1 # Always ready for RAM
   IandX = MRUMode2 | MRUAdr12-6
   FlashXAdr = IandX When ((¬XE) ∨ (SE ∧ OE)) Else XadrReg
  FlashAdr = FlashXAdr | MRUAdr5-0 # Merge X and Y addresses
   XadrCh = 1 When ((XadrReg /= IandX) ∧ XE ∧(¬SE) ∧(¬OE) ∧FlashNewTrans) Else 0
# Xadr change
  RAMAdr = MRUAdr9-3 When IsValidAccess # Maximum address = 95
           Else 0
   RAMWE = 0 When IsValidAccess ∧¬ MRUModeint0
   MRUModeint = MRUMode1-0 # Backwards compatability


  FlashCtrl(0) = IFREN
  FlashCtrl(1) = XE
  FlashCtrl(2) = YE
  FlashCtrl(3) = SE
  FlashCtrl(4) = OE
  FlashCtrl(5) = PROG
  FlashCtrl(6) = NVSTR
  FlashCtrl(7) = ERASE
  FlashCtrl(8) = MAS1

 MemClk       = ¬Clk # Memory clock
```

REFERENCES

20 References

[1] Silverbrook Research, 1998, *4-3-1-3 Authentication of Consumables.*
[2] Silverbrook Research, 2002, *4-3-1-26 Authentication Protocols.*
[3] Silverbrook Research, 2002, *4-3-1-8 QID Requirements Specification.*
[4] TSMC, Oct 1, 2000, *SFC0008_08B9_HE*, 8K × 8 Embedded Flash Memory Specification, Rev 0.1.
[5] TSMC (design service division), Sep 10, 2001, *0.25um Embedded Flash Test Mode User Guide*, V0.3.
[6] TSMC (EmbFlash product marketing), Oct 19, 2001, *0.25um Application Note*, V2.2.

Authentication of Consumables

version 1.4

Silverbrook Research Pty Ltd
393 Darling Street, Balmain
NSW 2041 Australia
Phone: +61 2 9818 6633
Fax: +61 2 9818 6711
Email: info@silverbrook.com.au

Document History

Version	Date	Author	Details
1.4	25 November 2002	Simon Walmsley	Moved theory document back here (4-3-1-3) after it had spent some time inside the chip implementation document (4-3-1-1). Note that although this document was published November 2002, it has not been updated for content since 1998. While the background is still relevant, the protocols are not up to date. For more updated information about security and authentication protocols, see 4-3-1-26 and 4-4-1-3.
1.3	7 December, 1998	Simon Walmsley	Converted to FrameMaker. Updated after review. Added SEPM, Trojan horse, and overwrite flash (electron beam) attacks. Added Protocols P3 and C3. Added checksum of keys in protocol C1 before each HMAC-SHA1 call is processed. More detail in discussions.
1.2	7 July, 1998	Simon Walmsley	Fixed LFSR taps in error.
1.1	1 July, 1998	Simon Walmsley	Split into 2 documents: This one (theory), and chip implementation.
1.0	5 June, 1998	Simon Walmsley	Initial issue.

Contents

INTRODUCTION

1 Background

Manufacturers of systems that require consumables (such as a laser printer that requires toner cartridges) have struggled with the problem of authenticating consumables, to varying levels of success. Most have resorted to specialized packaging that involves a patent. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. The prevention of copying is important for two reasons:

- To protect revenues
- To prevent poorly manufactured substitute consumables from damaging the base system. For example, poorly filtered ink may clog print nozzles in an ink jet printer, causing the consumer to blame the system manufacturer and not admit the use of non-authorized consumables.

To solve the authentication problem, this document describes an QA Chip that contains authentication keys and circuitry specially designed to prevent copying. The chip is manufactured using the standard Flash memory manufacturing process, and is low cost enough to be included in consumables such as ink and toner cartridges. The implementation is approximately $1mm^2$ in a 0.25 micron flash process, and has an expected manufacturing cost of approximately 10 cents in 2003.

Once programmed, the QA Chips as described here are compliant with the NSA export guidelines since they do not constitute a strong encryption device. They can therefore be practically manufactured in the USA (and exported) or anywhere else in the world.

This reference describes the theory behind the authentication mechanism used in the authentication chips, and the authentication protocols used by embedded systems containing this chip. Companion documents [84] and [86] describe the implementation of an authentication chip, and an authentication chip testing / programming device respectively. Companion documents [85] and [87] describe updated protocols made after this document.

2 Readership

This document is written for hardware engineers, software engineers, production engineers and system architects.

This document is confidential to Silverbrook Research Pty. Ltd. and its distribution outside this organisation *must* be covered by a non-disclosure agreement (NDA).

3 Warning

Note that although this document was published November 2002, it has not been updated for content since 1998. While the background section remains relevant, the protocols are not up to date. For more updated information about security and authentication protocols, see [85] and [87].

4 Nomenclature

The following symbolic nomenclature is used throughout this document:

Table 1. Summary of symbolic nomenclature

Symbol	Description
F[X]	Function F, taking a single parameter X
F[X, Y]	Function F, taking two parameters, X and Y
X | Y	X concatenated with Y
X ∧ Y	Bitwise X AND Y
X ∨ Y	Bitwise X OR Y (inclusive-OR)
X ⊕ Y	Bitwise X XOR Y (exclusive-OR)
¬X	Bitwise NOT X (complement)
X ← Y	X is assigned the value Y
X ← {Y, Z}	The domain of assignment inputs to X is Y and Z
X = Y	X is equal to Y
X ≠ Y	X is not equal to Y
⇩X	Decrement X by 1 (floor 0)
⇧X	Increment X by 1 (modulo register length)
Erase X	Erase Flash memory register X
SetBits[X, Y]	Set the bits of the Flash memory register X based on Y
Z ← ShiftRight[X, Y]	Shift register X right one bit position, taking input bit from Y and placing the output bit in Z

4.1 PSEUDOCODE

4.1.1 Asynchronous

The following pseudocode:

```
var = expression
```

means the var signal or output is equal to the evaluation of the expression.

4.1.2 Synchronous

The following pseudocode:

```
var ← expression
```

means the var register is assigned the result of evaluating the expression during this cycle.

4.1.3 Expression

Expressions are defined using the nomenclature in Table 1 above. Therefore:

```
var = (a = b)
```

is interpreted as the var signal is 1 if a is equal to b, and 0 otherwise.

4.2 DIAGRAMS

Black is used to denote data, and red to denote 1-bit control-signal lines.

4.3 QA CHIP TERMINOLOGY

This document refers to QA Chips by their function in particular protocols:

- For authenticated reads, ChipA is the QA Chip being authenticated, and ChipT is the QA Chip that is trusted.
- For replacement of keys, ChipP is the QA Chip being programmed with the new key, and ChipF is the factory QA Chip that generates the message to program the new key.
- For upgrades of data in a QA Chip, ChipU is the QA Chip being upgraded, and ChipS is the QA Chip that signs the upgrade value.

Any given physical QA Chip will contain functionality that allows it to operate as an entity in some number of these protocols.

Therefore, wherever the terms ChipA, ChipT, ChipP, ChipF, ChipU and ChipS are used in this document, they are referring to *logical* entities involved in an authentication protocol as defined in subsequent sections.

Physical QA Chips are referred to by their location. For example, each ink cartridge may contain a QA Chip referred to as an INK_QA, with all INK_QA chips being on the same physical bus. In the same way, the QA Chip inside a printer is referred to as PRINTER_QA, and will be on a separate bus to the INK_QA chips.

5 Concepts and Terms

This chapter provides a background to the problem of authenticating consumables. For more in-depth introductory texts, see [12], [78], and [56].

5.1 BASIC TERMS

A message, denoted by M, is ***plaintext***. The process of transforming M into ***ciphertext*** C, where the substance of M is hidden, is called ***encryption***. The process of transforming C back into M is called ***decryption***. Referring to the encryption function as **E**, and the decryption function as **D**, we have the following identities:

$$E[M] = C$$
$$D[C] = M$$

Therefore the following identity is true: $D[E[M]] = M$

5.2 SYMMETRIC CRYPTOGRAPHY

A symmetric encryption algorithm is one where:

- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 can be derived from K_1, and
- K_1 can be derived from K_2.

In most symmetric algorithms, K_1 equals K_2. However, even if K_1 does not equal K_2, given that one key can be derived from the other, a single key K can suffice for the mathematical definition. Thus:

$$E_K[M] = C$$
$$D_K[C] = M$$

The security of these algorithms rests very much in the key K. Knowledge of K allows *anyone* to encrypt or decrypt. Consequently K must remain a secret for the duration of the value of M. For example, M may be a wartime message "My current position is grid position 123-456". Once the war is over the value of M is greatly reduced, and if K is made public, the knowledge of the combat unit's position may be of no relevance whatsoever. Of course if it is politically sensitive for the combat unit's position to be known even after the war, K may have to remain secret for a very long time.

An enormous variety of symmetric algorithms exist, from the textbooks of ancient history through to sophisticated modern algorithms. Many of these are insecure, in that modern cryptanalysis techniques (see Section 5.7 on page 18) can successfully attack the algorithm to the extent that K can be derived.

The security of the particular symmetric algorithm is a function of two things: the strength of the algorithm and the length of the key [78].

The strength of an algorithm is difficult to quantify, relying on its resistance to cryptographic attacks (see Section 5.7 on page 18). In addition, the longer that an algorithm has remained in the public eye, and yet remained unbroken in the midst of intense scrutiny, the

more secure the algorithm is likely to be. By contrast, a secret algorithm that has not been scrutinized by cryptographic experts is unlikely to be secure.

Even if the algorithm is "perfectly" strong (the only way to break it is to try every key - see Section 5.7.1.5 on page 19), eventually the right key will be found. However, the more keys there are, the more keys have to be tried. If there are N keys, it will take a maximum of N tries. If the key is N bits long, it will take a maximum of 2^N tries, with a 50% chance of finding the key after only half the attempts (2^{N-1}). The longer N becomes, the longer it will take to find the key, and hence the more secure it is. What makes a good key length depends on the value of the secret and the time for which the secret must remain secret as well as available computing resources.

In 1996, an ad hoc group of world-renowned cryptographers and computer scientists released a report [9] describing minimal key lengths for symmetric ciphers to provide adequate commercial security. They suggest an absolute minimum key length of 90 bits in order to protect data for 20 years, and stress that increasingly, as cryptosystems succumb to smarter attacks than brute-force key search, even more bits may be required to account for future surprises in cryptanalysis techniques.

We will ignore most historical symmetric algorithms on the grounds that they are insecure, especially given modern computing technology. Instead, we will discuss the following algorithms:

- DES
- Blowfish
- RC5
- IDEA

5.2.1 DES

DES (Data Encryption Standard) [26] is a US and international standard, where the same key is used to encrypt and decrypt. The key length is 56 bits. It has been implemented in hardware and software, although the original design was for hardware only. The original algorithm used in DES was patented in 1976 (US patent number 3,962,539) and has since expired.

During the design of DES, the NSA (National Security Agency) provided secret S-boxes to perform the key-dependent nonlinear transformations of the data block. After differential cryptanalysis was discovered outside the NSA, it was revealed that the DES S-boxes were specifically designed to be resistant to differential cryptanalysis.

As described in [95], using 1993 technology, a 56-bit DES key can be recovered by a custom-designed $1 million machine performing a brute force attack in only 35 minutes. For $10 million, the key can be recovered in only 3.5 minutes. DES is clearly not secure now, and will become less so in the future.

A variant of DES, called *triple-DES* is more secure, but requires 3 keys: K_1, K_2, and K_3. The keys are used in the following manner:

$$E_{K3}[D_{K2}[E_{K1}[M]]] = C$$
$$D_{K3}[E_{K2}[D_{K1}[C]]] = M$$

The main advantage of triple-DES is that existing DES implementations can be used to give more security than single key DES. Specifically, triple-DES gives protection of

equivalent key length of 112 bits [78]. Triple-DES does not give the equivalent protection of a 168-bit key (3 × 56) as one might naively expect.

Equipment that performs triple-DES decoding and/or encoding cannot be exported from the United States.

5.2.2 Blowfish

Blowfish is a symmetric block cipher first presented by Schneier in 1994 [76]. It takes a variable length key, from 32 bits to 448 bits, is unpatented, and is both license and royalty free. In addition, it is much faster than DES.

The Blowfish algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key of at most 448 bits into several subkey arrays totaling 4168 bytes. Data encryption occurs via a 16-round Feistel network. All operations are XORs and additions on 32-bit words, with four index array lookups per round.

It should be noted that decryption is the same as encryption except that the subkey arrays are used in the reverse order. Complexity of implementation is therefore reduced compared to other algorithms that do not have such symmetry.

[77] describes the published attacks which have been mounted on Blowfish, although the algorithm remains secure as of February 1998 [79]. The major finding with these attacks has been the discovery of certain weak keys. These weak keys can be tested for during key generation. For more information, refer to [77] and [79].

5.2.3 RC5

Designed by Ron Rivest in 1995, RC5 [74] has a variable block size, key size, and number of rounds. Typically, however, it uses a 64-bit block size and a 128-bit key.

The RC5 algorithm consists of two parts: a key-expansion part and a data-encryption part. Key expansion converts a key into $2r+2$ subkeys (where r = the number of rounds), each subkey being w bits. For a 64-bit blocksize with 16 rounds (w=32, r=16), the subkey arrays total 136 bytes. Data encryption uses addition mod 2^w, XOR and bitwise rotation.

An initial examination by Kaliski and Yin [43] suggested that standard linear and differential cryptanalysis appeared impractical for the 64-bit blocksize version of the algorithm. Their differential attacks on 9 and 12 round RC5 require 2^{45} and 2^{62} chosen plaintexts respectively, while the linear attacks on 4, 5, and 6 round RC5 requires 2^{37}, 2^{47} and 2^{57} known plaintexts). These two attacks are independent of key size.

More recently however, Knudsen and Meier [47] described a new type of differential attack on RC5 that improved the earlier results by a factor of 128, showing that RC5 has certain weak keys.

RC5 is protected by multiple patents owned by RSA Laboratories. A license must be obtained to use it.

5.2.4 IDEA

Developed in 1990 by Lai and Massey [53], the first incarnation of the IDEA cipher was called PES. After differential cryptanalysis was discovered by Biham and Shamir in 1991, the algorithm was strengthened, with the result being published in 1992 as IDEA [52].

IDEA uses 128-bit keys to operate on 64-bit plaintext blocks. The same algorithm is used for encryption and decryption. It is generally regarded as the most secure block algorithm available today [78][56].

The biggest drawback of IDEA is the fact that it is patented (US patent number 5,214,703, issued in 1993), and a license must be obtained from Ascom Tech AG (Bern) to use it.

5.3 ASYMMETRIC CRYPTOGRAPHY

An asymmetric encryption algorithm is one where:

- the encryption function E relies on key K_1,
- the decryption function D relies on key K_2,
- K_2 cannot be derived from K_1 in a reasonable amount of time, and
- K_1 cannot be derived from K_2 in a reasonable amount of time.

Thus:
$$E_{K1}[M] = C$$
$$D_{K2}[C] = M$$

These algorithms are also called ***public-key*** because one key K_1 can be made public. Thus anyone can encrypt a message (using K_1) but only the person with the corresponding decryption key (K_2) can decrypt and thus read the message.

In most cases, the following identity also holds:
$$E_{K2}[M] = C$$
$$D_{K1}[C] = M$$

This identity is very important because it implies that anyone with the public key K_1 can see M and know that it came from the owner of K_2. No-one else could have generated C because to do so would imply knowledge of K_2. This gives rise to a different application, unrelated to encryption - digital signatures.

The property of not being able to derive K_1 from K_2 and vice versa in a reasonable time is of course clouded by the concept of *reasonable time*. What has been demonstrated time after time, is that a calculation that was thought to require a long time has been made possible by the introduction of faster computers, new algorithms etc. The security of asymmetric algorithms is based on the difficulty of one of two problems: factoring large numbers (more specifically large numbers that are the product of two large primes), and the difficulty of calculating discrete logarithms in a finite field. Factoring large numbers is conjectured to be a hard problem given today's understanding of mathematics. The problem however, is that factoring is getting easier much faster than anticipated. Ron Rivest in 1977 said that factoring a 125-digit number would take 40 quadrillion years [30]. In 1994 a 129-digit number was factored [3]. According to Schneier, you need a 1024-bit number to get the level of security today that you got from a 512-bit number in the 1980s [78]. If the key is to last for some years then 1024 bits may not even be enough. Rivest revised his key length estimates in 1990: he suggests 1628 bits for high security lasting until 2005, and 1884 bits for high security lasting until 2015 [69]. Schneier suggests 2048 bits are required in order to protect against corporations and governments until 2015 [80].

Public key cryptography was invented in 1976 by Diffie and Hellman [15][16], and independently by Merkle [57]. Although Diffie, Hellman and Merkle patented the concepts (US patent numbers 4,200,770 and 4,218,582), these patents expired in 1997.

A number of public key cryptographic algorithms exist. Most are impractical to implement, and many generate a very large C for a given M or require enormous keys. Still others, while secure, are far too slow to be practical for several years. Because of this, many

public key systems are hybrid - a public key mechanism is used to transmit a symmetric session key, and then the session key is used for the actual messages.

All of the algorithms have a problem in terms of key selection. A random number is simply not secure enough. The two large primes p and q must be chosen carefully - there are certain weak combinations that can be factored more easily (some of the weak keys can be tested for). But nonetheless, key selection is not a simple matter of randomly selecting 1024 bits for example. Consequently the key selection process must also be secure.

Of the practical algorithms in use under public scrutiny, the following are discussed:

- RSA
- DSA
- ElGamal

5.3.1 RSA

The RSA cryptosystem [75], named after Rivest, Shamir, and Adleman, is the most widely used public key cryptosystem, and is a de facto standard in much of the world [78].

The security of RSA depends on the conjectured difficulty of factoring large numbers that are the product of two primes (p and q). There are a number of restrictions on the generation of p and q. They should both be large, with a similar number of bits, yet not be close to one another (otherwise $p \equiv q \equiv \sqrt{pq}$). In addition, many authors have suggested that p and q should be strong primes [56]. The Hellman-Bach patent (US patent number 4,633,036) covers a method for generating strong RSA primes p and q such that $n = pq$ and factoring n is believed to be computationally infeasible.

The RSA algorithm patent was issued in 1983 (US patent number 4,405,829). The patent expires on September 20, 2000.

5.3.2 DSA

DSA (Digital Signature Algorithm) is an algorithm designed as part of the Digital Signature Standard (DSS) [29]. As defined, it cannot be used for generalized encryption. In addition, compared to RSA, DSA is 10 to 40 times slower for signature verification [40]. DSA explicitly uses the SHA-1 hashing algorithm (see Section 5.5.3.3 on page 14).

DSA key generation relies on finding two primes p and q such that q divides p-1. According to Schneier [78], a 1024-bit p value is required for long term DSA security. However the DSA standard [29] does not permit values of p larger than 1024 bits (p must also be a multiple of 64 bits).

The US Government owns the DSA algorithm and has at least one relevant patent (US patent 5,231,688 granted in 1993). However, according to NIST [61]:

> *"The DSA patent and any foreign counterparts that may issue are available for use without any written permission from or any payment of royalties to the U.S. government."*

In a much stronger declaration, NIST states in the same document [61] that DSA does not infringe third party's rights:

> *"NIST reviewed all of the asserted patents and concluded that none of them would be infringed by DSS. Extra protection will be written into the PKI pilot project that will prevent an organization or individual*

from suing anyone except the government for patent infringement during the course of the project."

It must however, be noted that the Schnorr authentication algorithm [81] (US patent 4,995,082) patent holder claims that DSA infringes his patent. The Schnorr patent is not due to expire until 2008.

5.3.3 ElGamal

The ElGamal scheme [22][23] is used for both encryption and digital signatures. The security is based on the conjectured difficulty of calculating discrete logarithms in a finite field.

Key selection involves the selection of a prime p, and two random numbers g and x such that both g and x are less than p. Then calculate $y = gx \bmod p$. The public key is y, g, and p. The private key is x.

ElGamal is unpatented. Although it uses the patented Diffie-Hellman public key algorithm [15][16], those patents expired in 1997. ElGamal public key encryption and digital signatures can now be safely used without infringing third party patents.

5.4 CRYPTOGRAPHIC CHALLENGE-RESPONSE PROTOCOLS AND ZERO KNOWLEDGE PROOFS

The general principle of a challenge-response protocol is to provide identity authentication. The simplest form of challenge-response takes the form of a secret password. A asks B for the secret password, and if B responds with the correct password, A declares B authentic.

There are three main problems with this kind of simplistic protocol. Firstly, once B has responded with the password, any observer C will know what the password is. Secondly, A must know the password in order to verify it. Thirdly, if C impersonates A, then B will give the password to C (thinking C was A), thus compromising the password.

Using a copyright text (such as a *haiku*) as the password is not sufficient, because we are assuming that anyone is able to copy the password (for example in a country where intellectual property is not respected).

The idea of *cryptographic challenge-response protocols* is that one entity (the claimant) proves its identity to another (the verifier) by demonstrating knowledge of a secret known to be associated with that entity, *without revealing the secret itself* to the verifier during the protocol [56]. In the generalized case of cryptographic challenge-response protocols, with some schemes the verifier knows the secret, while in others the secret is not even known by the verifier. A good overview of these protocols can be found in [25], [78], and [56].

Since this documentation specifically concerns Authentication, the actual cryptographic challenge-response protocols used for authentication are detailed in the appropriate sections. However the concept of Zero Knowledge Proofs bears mentioning here.

The Zero Knowledge Proof protocol, first described by Feige, Fiat and Shamir in [24] is extensively used in Smart Cards for the purpose of authentication [34][36][67]. The protocol's effectiveness is based on the assumption that it is computationally infeasible to com-

pute square roots modulo a large composite integer with unknown factorization. This is provably equivalent to the assumption that factoring large integers is difficult.

It should be noted that there is no need for the claimant to have significant computing power. Smart cards implement this kind of authentication using only a few modulo multiplications [34][36].

Finally, it should be noted that the Zero Knowledge Proof protocol is patented [82] (US patent 4,748,668, issued May 31, 1988).

5.5 ONE-WAY FUNCTIONS

A one-way function F operates on an input X, and returns F[X] such that X cannot be determined from F[X]. When there is no restriction on the format of X, and F[X] contains fewer bits than X, then collisions must exist. A collision is defined as two different X input values producing the same F[X] value - i.e. X_1 and X_2 exist such that $X_1 \neq X_2$ yet $F[X_1] = F[X_2]$.

When X contains more bits than F[X], the input must be compressed in some way to create the output. In many cases, X is broken into blocks of a particular size, and compressed over a number of rounds, with the output of one round being the input to the next. The output of the hash function is the last output once X has been consumed. A *pseudo-collision* of the compression function CF is defined as two different initial values V_1 and V_2 and two inputs X_1 and X_2 (possibly identical) are given such that $CF(V_1, X_1) = CF(V_2, X_2)$. Note that the existence of a pseudo-collision does not mean that it is easy to compute an X_2 for a given X_1.

We are only interested in one-way functions that are fast to compute. In addition, we are only interested in *deterministic* one-way functions that are repeatable in different implementations. Consider an example F where F[X] is the time between calls to F. For a given F[X] X cannot be determined because X is not even used by F. However the output from F will be different for different implementations. This kind of F is therefore not of interest.

In the scope of this document, we are interested in the following forms of one-way functions:

- Encryption using an unknown key
- Random number sequences
- Hash Functions
- Message Authentication Codes

5.5.1 Encryption using an unknown key

When a message is encrypted using an unknown key K, the encryption function E is effectively one-way. Without the key, it is computationally infeasible to obtain M from EK[M] without K. An encryption function is only one-way for as long as the key remains hidden.

An encryption algorithm does not create collisions, since E creates EK[M] such that it is possible to reconstruct M using function D. Consequently F[X] contains at least as many bits as X (no information is lost) if the one-way function F is E.

Symmetric encryption algorithms (see Section 5.2 on page 5) have the advantage over asymmetric algorithms (see Section 5.3 on page 8) for producing one-way functions based on encryption for the following reasons:

- The key for a given strength encryption algorithm is shorter for a symmetric algorithm than an asymmetric algorithm
- Symmetric algorithms are faster to compute and require less software or silicon

Note however, that the selection of a good key depends on the encryption algorithm chosen. Certain keys are not strong for particular encryption algorithms, so any key needs to be tested for strength. The more tests that need to be performed for key selection, the less likely the key will remain hidden.

5.5.2 Random number sequences

Consider a random number sequence R_0, R_1, ..., R_i, R_{i+1}. We define the one-way function F such that F[X] returns the X^{th} random number in the random sequence. However we must ensure that F[X] is repeatable for a given X on different implementations. The random number sequence therefore cannot be truly random. Instead, it must be pseudo-random, with the generator making use of a specific seed.

There are a large number of issues concerned with defining good random number generators. Knuth, in [48] describes what makes a generator "good" (including statistical tests), and the general problems associated with constructing them. Moreau gives a high level survey of the current state of the field in [60].

The majority of random number generators produce the i^{th} random number from the $i\text{-}1^{th}$ state - the only way to determine the i^{th} number is to iterate from the 0^{th} number to the i^{th}. If i is large, it may not be practical to wait for i iterations.

However there is a type of random number generator that *does* allow random access. In [10], Blum, Blum and Shub define the ideal generator as follows: *"... we would like a pseudo-random sequence generator to quickly produce, from short seeds, long sequences (of bits) that appear in every way to be generated by successive flips of a fair coin"*. They defined the $x^2 \bmod n$ generator [10], more commonly referred to as the BBS generator. They showed that given certain assumptions upon which modern cryptography relies, a BBS generator passes extremely stringent statistical tests.

The BBS generator relies on selecting n which is a Blum integer ($n = pq$ where p and q are large prime numbers, $p \neq q$, $p \bmod 4 = 3$, and $q \bmod 4 = 3$). The initial state of the generator is given by x_0 where $x_0 = x^2 \bmod n$, and x is a random integer relatively prime to n. The i^{th} pseudo-random bit is the least significant bit of x_i where:

$$x_i = x_{i-1}^2 \bmod n$$

As an extra property, knowledge of p and q allows a direct calculation of the i^{th} number in the sequence as follows:

$$x_i = x_0^{\,y} \bmod n \qquad \text{where } y = 2^i \bmod ((p-1)(q-1))$$

Without knowledge of p and q, the generator must iterate (the security of calculation relies on the conjectured difficulty of factoring large numbers).

When first defined, the primary problem with the BBS generator was the amount of work required for a single output bit. The algorithm was considered too slow for most applications. However the advent of Montgomery reduction arithmetic [58] has given rise to more practical implementations, such as [59]. In addition, Vazirani and Vazirani have

shown in [93] that depending on the size of n, more bits can safely be taken from x_i without compromising the security of the generator.

Assuming we only take 1 bit per x_i, N bits (and hence N iterations of the bit generator function) are needed in order to generate an N-bit random number. To the outside observer, given a particular set of bits, there is no way to determine the next bit other than a 50/50 probability. If the x, p and q are hidden, they act as a key, and it is computationally infeasible to take an output bit stream and compute x, p, and q. It is also computationally infeasible to determine the value of i used to generate a given set of pseudo-random bits. This last feature makes the generator one-way. Different values of i can produce identical bit sequences of a given length (e.g. 32 bits of random bits). Even if x, p and q are known, for a given $F[i]$, i can only be derived as a set of possibilities, not as a certain value (of course if the domain of i is known, then the set of possibilities is reduced further).

However, there are problems in selecting a good p and q, and a good seed x. In particular, Ritter in [68] describes a problem in selecting x. The nature of the problem is that a BBS generator does not create a single cycle of known length. Instead, it creates cycles of various lengths, including degenerate (zero-length) cycles. Thus a BBS generator cannot be initialized with a random state - it might be on a short cycle. Specific algorithms exist in section 9 of [10] to determine the length of the period for a given seed given certain strenuous conditions for n.

5.5.3 Hash functions

Special one-way functions, known as Hash functions, map arbitrary length messages to fixed-length hash values. Hash functions are referred to as H[M]. Since the input is of arbitrary length, a hash function has a compression component in order to produce a fixed length output. Hash functions also have an obfuscation component in order to make it difficult to find collisions and to determine information about M from H[M].

Because collisions do exist, most applications require that the hash algorithm is preimage resistant, in that for a given X_1 it is difficult to find X_2 such that $H[X_1] = H[X_2]$. In addition, most applications also require the hash algorithm to be *collision resistant* (i.e. it should be hard to find two messages X_1 and X_2 such that $H[X_1] = H[X_2]$). However, as described in [20], it is an open problem whether a collision-resistant hash function, in the ideal sense, can exist at all.

The primary application for hash functions is in the reduction of an input message into a digital "fingerprint" before the application of a digital signature algorithm. One problem of collisions with digital signatures can be seen in the following example.

> A has a long message M_1 that says *"I owe B $10"*. A signs $H[M_1]$ using his private key. B, being greedy, then searches for a collision message M_2 where $H[M_2] = H[M_1]$ but where M_2 is favorable to B, for example *"I owe B $1million"*. Clearly it is in A's interest to ensure that it is difficult to find such an M_2.

Examples of collision resistant one-way hash functions are SHA-1 [28], MD5 [73] and RIPEMD-160 [66], all derived from MD4 [70][72].

5.5.3.1 MD4

Ron Rivest introduced MD4 [70][72] in 1990. It is only mentioned here because all other one-way hash functions are derived in some way from MD4.

MD4 is now considered completely broken [18][19] in that collisions can be calculated instead of searched for. In the example above, B could trivially generate a substitute message M_2 with the same hash value as the original message M_1.

5.5.3.2 MD5

Ron Rivest introduced MD5 [73] in 1991 as a more secure MD4. Like MD4, MD5 produces a 128-bit hash value. MD5 is not patented [80].

Dobbertin describes the status of MD5 after recent attacks [20]. He describes how pseudo-collisions have been found in MD5, indicating a weakness in the compression function, and more recently, collisions have been found. This means that MD5 should not be used for compression in digital signature schemes where the existence of collisions may have dire consequences. However MD5 can still be used as a one-way function. In addition, the HMAC-MD5 construct (see Section 5.5.4.1 on page 16) is not affected by these recent attacks.

5.5.3.3 SHA-1

SHA-1 [28] is very similar to MD5, but has a 160-bit hash value (MD5 only has 128 bits of hash value). SHA-1 was designed and introduced by the NIST and NSA for use in the Digital Signature Standard (DSS). The original published description was called SHA [27], but very soon afterwards, was revised to become SHA-1 [28], supposedly to correct a security flaw in SHA (although the NSA has not released the mathematical reasoning behind the change).

There are no known cryptographic attacks against SHA-1 [78]. It is also more resistant to brute force attacks than MD4 or MD5 simply because of the longer hash result.

The US Government owns the SHA-1 and DSA algorithms (a digital signature authentication algorithm defined as part of DSS [29]) and has at least one relevant patent (US patent 5,231,688 granted in 1993). However, according to NIST [61]:

> *"The DSA patent and any foreign counterparts that may issue are available for use without any written permission from or any payment of royalties to the U.S. government."*

In a much stronger declaration, NIST states in the same document [61] that DSA and SHA-1 do not infringe third party's rights:

> *"NIST reviewed all of the asserted patents and concluded that none of them would be infringed by DSS. Extra protection will be written into the PKI pilot project that will prevent an organization or individual from suing anyone except the government for patent infringement during the course of the project."*

It must however, be noted that the Schnorr authentication algorithm [81] (US patent number 4,995,082) patent holder claims that DSA infringes his patent. The Schnorr patent is not due to expire until 2008. Fortunately this does not affect SHA-1.

5.5.3.4 RIPEMD-160

RIPEMD-160 [66] is a hash function derived from its predecessor RIPEMD [11] (developed for the European Community's RIPE project in 1992). As its name suggests,

RIPEMD-160 produces a 160-bit hash result. Tuned for software implementations on 32-bit architectures, RIPEMD-160 is intended to provide a high level of security for 10 years or more.

Although there have been no successful attacks on RIPEMD-160, it is comparatively new and has not been extensively cryptanalyzed. The original RIPEMD algorithm [11] was specifically designed to resist known cryptographic attacks on MD4. The recent attacks on MD5 (detailed in [20]) showed similar weaknesses in the RIPEMD 128-bit hash function. Although the attacks showed only theoretical weaknesses, Dobbertin, Preneel and Bosselaers further strengthened RIPEMD into a new algorithm RIPEMD-160.

RIPEMD-160 is in the public domain, and requires no licensing or royalty payments.

5.5.4 Message authentication codes

The problem of message authentication can be summed up as follows:

How can A be sure that a message supposedly from B is in fact from B?

Message authentication is different from entity authentication (described in the section on cryptographic challenge-response protocols). With entity authentication, one entity (the claimant) proves its identity to another (the verifier). With message authentication, we are concerned with making sure that a given message is from who we think it is from i.e. it has not been tampered with en route from the source to its destination. While this section has a brief overview of message authentication, a more detailed survey can be found in [88].

A one-way hash function is not sufficient protection for a message. Hash functions such as MD5 rely on generating a hash value that is representative of the original input, and the original input cannot be derived from the hash value. A simple attack by E, who is in-between A and B, is to intercept the message from B, and substitute his own. Even if A also sends a hash of the original message, E can simply substitute the hash of his new message. Using a one-way hash function alone, A has no way of knowing that B's message has been changed.

One solution to the problem of message authentication is the Message Authentication Code, or MAC.

When B sends message M, it also sends MAC[M] so that the receiver will know that M is actually from B. For this to be possible, only B must be able to produce a MAC of M, and in addition, A should be able to verify M against MAC[M]. Notice that this is different from encryption of M - MACs are useful when M does not have to be secret.

The simplest method of constructing a MAC from a hash function is to encrypt the hash value with a symmetric algorithm:

1. Hash the input message H[M]
2. Encrypt the hash $E_K[H[M]]$

This is more secure than first encrypting the message and then hashing the encrypted message. Any symmetric or asymmetric cryptographic function can be used, with the appropriate advantages and disadvantage of each type described in Section 5.2 on page 5 and Section 5.3 on page 8.

However, there are advantages to using a *key-dependent one-way hash function* instead of techniques that use encryption (such as that shown above):

- Speed, because one-way hash functions in general work much faster than encryption;
- Message size, because $E_K[M]$ is at least the same size as M, while H[M] is a fixed size (usually considerably smaller than M);
- Hardware/software requirements - keyed one-way hash functions are typically far less complex than their encryption-based counterparts; and
- One-way hash function implementations are not considered to be encryption or decryption devices and therefore are not subject to US export controls.

It should be noted that hash functions were never originally designed to contain a key or to support message authentication. As a result, some ad hoc methods of using hash functions to perform message authentication, including various functions that concatenate messages with secret prefixes, suffixes, or both have been proposed [56][78]. Most of these ad hoc methods have been successfully attacked by sophisticated means [42][64][65]. Additional MACs have been suggested based on XOR schemes [8] and Toeplitz matrices [49] (including the special case of LFSR-based (Linear Feed Shift Register) constructions).

5.5.4.1 HMAC

The HMAC construction [6][7] in particular is gaining acceptance as a solution for Internet message authentication security protocols. The HMAC construction acts as a wrapper, using the underlying hash function in a black-box way. Replacement of the hash function is straightforward if desired due to security or performance reasons. However, the major advantage of the HMAC construct is that it can be proven secure provided the underlying hash function has some reasonable cryptographic strengths - that is, HMAC's strengths are directly connected to the strength of the hash function [6].

Since the HMAC construct is a wrapper, any iterative hash function can be used in an HMAC. Examples include HMAC-MD5, HMAC-SHA1, HMAC-RIPEMD160 etc.

Given the following definitions:

- H = the hash function (e.g. MD5 or SHA-1)
- n = number of bits output from H (e.g. 160 for SHA-1, 128 bits for MD5)
- M = the data to which the MAC function is to be applied
- K = the secret key shared by the two parties
- ipad = 0x36 repeated 64 times
- opad = 0x5C repeated 64 times

The HMAC algorithm is as follows:

1. Extend K to 64 bytes by appending 0x00 bytes to the end of K
2. XOR the 64 byte string created in (1) with ipad
3. append data stream M to the 64 byte string created in (2)
4. Apply H to the stream generated in (3)
5. XOR the 64 byte string created in (1) with opad
6. Append the H result from (4) to the 64 byte string resulting from (5)
7. Apply H to the output of (6) and output the result

Thus:

$$HMAC[M] = H[(K \oplus opad) \mid H[(K \oplus ipad) \mid M]]$$

The recommended key length is at least *n* bits, although it should not be longer than 64 bytes (the length of the hashing block). A key longer than *n* bits does not add to the security of the function.

HMAC optionally allows truncation of the final output e.g. truncation to 128 bits from 160 bits.

The HMAC designers' Request for Comments [51] was issued in 1997, one year after the algorithm was first introduced. The designers claimed that the strongest known attack against HMAC is based on the frequency of collisions for the hash function H (see Section 14.10 on page 68), and is totally impractical for minimally reasonable hash functions:

> *As an example, if we consider a hash function like MD5 where the output length is 128 bits, the attacker needs to acquire the correct message authentication tags computed (with the same secret key K) on about 2^{64} known plaintexts. This would require the processing of at least 2^{64} blocks under H, an impossible task in any realistic scenario (for a block length of 64 bytes this would take 250,000 years in a continuous 1 Gbps link, and without changing the secret key K all this time). This attack could become realistic only if serious flaws in the collision behavior of the function H are discovered (e.g. Collisions found after 2^{30} messages). Such a discovery would determine the immediate replacement of function H (the effects of such a failure would be far more severe for the traditional uses of H in the context of digital signatures, public key certificates etc).*

Of course, if a 160-bit hash function is used, then 2^{64} should be replaced with 2^{80}.

This should be contrasted with a regular collision attack on cryptographic hash functions where no secret key is involved and 2^{64} off-line parallelizable operations suffice to find collisions.

More recently, HMAC protocols with replay prevention components [62] have been defined in order to prevent the capture and replay of any M, HMAC[M] combination within a given time period.

Finally, it should be noted that HMAC is in the public domain [50], and incurs no licensing fees. There are no known patents infringed by HMAC.

5.6 RANDOM NUMBERS AND TIME VARYING MESSAGES

The use of a random number generator as a one-way function has already been examined. However, random number generator theory is very much intertwined with cryptography, security, and authentication.

There are a large number of issues concerned with defining good random number generators. Knuth, in [48] describes what makes a generator good (including statistical tests), and the general problems associated with constructing them. Moreau gives a high level survey of the current state of the field in [60].

One of the uses for random numbers is to ensure that messages vary over time. Consider a system where A encrypts commands and sends them to B. If the encryption algorithm produces the same output for a given input, an attacker could simply record the messages and

play them back to fool B. There is no need for the attacker to crack the encryption mechanism other than to know which message to play to B (while pretending to be A). Consequently messages often include a random number and a time stamp to ensure that the message (and hence its encrypted counterpart) varies each time.

Random number generators are also often used to generate keys. Although Klapper has recently shown [45] that a family of secure feedback registers for the purposes of building key-streams *does* exist, he does not give any practical construction. It is therefore best to say at the moment that all generators are insecure for this purpose. For example, the Berlekamp-Massey algorithm [54], is a classic attack on an LFSR random number generator. If the LFSR is of length n, then only $2n$ bits of the sequence suffice to determine the LFSR, compromising the key generator.

If, however, the only role of the random number generator is to make sure that messages vary over time, the security of the generator and seed is not as important as it is for session key generation. If however, the random number seed generator is compromised, and an attacker is able to calculate future "random" numbers, it can leave some protocols open to attack. Any new protocol should be examined with respect to this situation.

The actual type of random number generator required will depend upon the implementation and the purposes for which the generator is used. Generators include Blum, Blum, and Shub [10], stream ciphers such as RC4 by Ron Rivest [71], hash functions such as SHA-1 [28] and RIPEMD-160 [66], and traditional generators such LFSRs (Linear Feedback Shift Registers) [48] and their more recent counterpart FCSRs (Feedback with Carry Shift Registers) [44].

5.7 ATTACKS

This section describes the various types of attacks that can be undertaken to break an authentication cryptosystem. The attacks are grouped into ***physical*** and ***logical*** attacks.

Logical attacks work on the protocols or algorithms rather than their physical implementation, and attempt to do one of three things:

- Bypass the authentication process altogether
- Obtain the secret key by force or deduction, so that *any* question can be answered
- Find enough about the nature of the authenticating questions and answers in order to, *without the key*, give the right answer to each question.

Regardless of the algorithms and protocol used by a security chip, the circuitry of the authentication part of the chip can come under physical attack. Physical attacks come in four main ways, although the form of the attack can vary:

- Bypassing the security chip altogether
- Physical examination of the chip while in operation (destructive and non-destructive)
- Physical decomposition of chip
- Physical alteration of chip

The attack styles and the forms they take are detailed below.

This section does not suggest solutions to these attacks. It merely describes each attack type. The examination is restricted to the context of an authentication chip (as opposed to some other kind of system, such as Internet authentication) attached to some System.

5.7.1 Logical attacks

These attacks are those which do not depend on the physical implementation of the cryptosystem. They work against the protocols and the security of the algorithms and random number generators.

5.7.1.1 *Ciphertext only attack*

This is where an attacker has one or more encrypted messages, all encrypted using the same algorithm. The aim of the attacker is to obtain the plaintext messages from the encrypted messages. Ideally, the key can be recovered so that all messages in the future can also be recovered.

5.7.1.2 *Known plaintext attack*

This is where an attacker has both the plaintext and the encrypted form of the plaintext. In the case of an authentication chip, a known-plaintext attack is one where the attacker can see the data flow between the system and the authentication chip. The inputs and outputs are observed (not chosen by the attacker), and can be analyzed for weaknesses (such as birthday attacks or by a search for differentially interesting input/output pairs).

A known plaintext attack can be carried out by connecting a logic analyzer to the connection between the system and the authentication chip.

5.7.1.3 *Chosen plaintext attacks*

A chosen plaintext attack describes one where a cryptanalyst has the ability to send any chosen message to the cryptosystem, and observe the response. If the cryptanalyst knows the algorithm, there may be a relationship between inputs and outputs that can be exploited by feeding a specific output to the input of another function.

The chosen plaintext attack is much stronger than the known plaintext attack since the attacker can choose the messages rather than simply observe the data flow.

On a system using an embedded authentication chip, it is generally very difficult to prevent chosen plaintext attacks since the cryptanalyst can logically pretend he/she is the system, and thus send any chosen bit-pattern streams to the authentication chip.

5.7.1.4 *Adaptive chosen plaintext attacks*

This type of attack is similar to the chosen plaintext attacks except that the attacker has the added ability to modify subsequent chosen plaintexts based upon the results of previous experiments. This is certainly the case with any system / authentication chip scenario described for consumables such as photocopiers and toner cartridges, especially since both systems and consumables are made available to the public.

5.7.1.5 *Brute force attack*

A *guaranteed* way to break *any* key-based cryptosystem algorithm is simply to try every key. Eventually the right one will be found. This is known as a ***brute force attack.*** However, the more key possibilities there are, the more keys must be tried, and hence the longer it takes (on average) to find the right one. If there are N keys, it will take a maximum of N tries. If the key is N bits long, it will take a maximum of 2^N tries, with a 50% chance of finding the key after only half the attempts (2^{N-1}). The longer N becomes, the longer it will take to find the key, and hence the more secure the key is. Of course, an attack may guess the key on the first try, but this is more unlikely the longer the key is.

Consider a key length of 56 bits. In the worst case, all 2^{56} tests (7.2×10^{16} tests) must be made to find the key. In 1977, Diffie and Hellman described a specialized machine for cracking DES, consisting of one million processors, each capable of running one million tests per second [17]. Such a machine would take 20 hours to break any DES code.

Consider a key length of 128 bits. In the worst case, all 2^{128} tests (3.4×10^{38} tests) must be made to find the key. This would take ten billion years on an array of a trillion processors each running 1 billion tests per second.

With a long enough key length, a brute force attack takes too long to be worth the attacker's efforts.

5.7.1.6 Guessing attack

This type of attack is where an attacker attempts to simply "guess" the key. As an attack it is identical to the brute force attack (see Section 5.7.1.5 on page 19) where the odds of success depend on the length of the key.

5.7.1.7 Quantum computer attack

To break an n-bit key, a quantum computer [83] (NMR, Optical, or Caged Atom) containing n qubits embedded in an appropriate algorithm must be built. The quantum computer effectively exists in 2^n simultaneous coherent states. The trick is to extract the right coherent state without causing any decoherence. To date this has been achieved with a 2 qubit system (which exists in 4 coherent states). It is thought possible to extend this to 6 qubits (with 64 simultaneous coherent states) within a few years.

Unfortunately, every additional qubit halves the relative strength of the signal representing the key. This rapidly becomes a serious impediment to key retrieval, especially with the long keys used in cryptographically secure systems.

As a result, attacks on a cryptographically secure key (e.g. 160 bits) using a Quantum Computer are likely not to be feasible and it is extremely unlikely that quantum computers will have achieved more than 50 or so qubits within the commercial lifetime of the authentication chips. Even using a 50 qubit quantum computer, 2^{110} tests are required to crack a 160 bit key.

5.7.1.8 Purposeful error attack

With certain algorithms, attackers can gather valuable information from the results of a bad input. This can range from the error message text to the time taken for the error to be generated.

A simple example is that of a userid/password scheme. If the error message usually says "Bad userid", then when an attacker gets a message saying "Bad password" instead, then they know that the userid is correct. If the message always says "Bad userid/password" then much less information is given to the attacker. A more complex example is that of the recent published method of cracking encryption codes from secure web sites [41]. The attack involves sending particular messages to a server and observing the error message responses. The responses give enough information to learn the keys - even the lack of a response gives some information.

An example of algorithmic time can be seen with an algorithm that returns an error as soon as an erroneous bit is detected in the input message. Depending on hardware imple-

mentation, it may be a simple method for the attacker to time the response and alter each bit one by one depending on the time taken for the error response, and thus obtain the key. Certainly in a chip implementation the time taken can be observed with far greater accuracy than over the Internet.

5.7.1.9 Birthday attack

This attack is named after the famous "birthday paradox" (which is not actually a paradox at all). The odds of one person sharing a birthday with another, is 1 in 365 (not counting leap years). Therefore there must be 183 people in a room for the odds to be more than 50% that one of them shares your birthday. However, there only needs to be 23 people in a room for there to be more than a 50% chance that any two share a birthday, as shown in the following relation:

$$Prob = 1 - \frac{nPr}{n^r} = 1 - \frac{365P23}{365^{23}} \approx 0.507$$

Birthday attacks are common attacks against hashing algorithms, especially those algorithms that combine hashing with digital signatures.

If a message has been generated and already signed, an attacker must search for a collision message that hashes to the same value (analogous to finding one person who shares your birthday). However, if the attacker can generate the message, the birthday attack comes into play. The attacker searches for two messages that share the same hash value (analogous to any two people sharing a birthday), only one message is acceptable to the person signing it, and the other is beneficial for the attacker. Once the person has signed the original message the attacker simply claims now that the person signed the alternative message - mathematically there is no way to tell which message was the original, since they both hash to the same value.

Assuming a brute force attack is the only way to determine a match, the weakening of an n-bit key by the birthday attack is $2^{n/2}$. A key length of 128 bits that is susceptible to the birthday attack has an effective length of only 64 bits.

5.7.1.10 Chaining attack

These are attacks made against the chaining nature of hash functions. They focus on the compression function of a hash function. The idea is based on the fact that a hash function generally takes arbitrary length input and produces a constant length output by processing the input n bits at a time. The output from one block is used as the chaining variable set into the next block. Rather than finding a collision against an entire input, the idea is that given an input chaining variable set, to find a substitute block that will result in the same output chaining variables as the proper message.

The number of choices for a particular block is based on the length of the block. If the chaining variable is c bits, the hashing function behaves like a random mapping, and the block length is b bits, the number of such b-bit blocks is approximately $2^b / 2^c$. The challenge for finding a substitution block is that such blocks are a sparse subset of all possible blocks.

For SHA-1, the number of 512 bit blocks is approximately $2^{512}/2^{160}$, or 2^{352}. The chance of finding a block by brute force search is about 1 in 2^{160}.

5.7.1.11 Substitution with a complete lookup table

If the number of potential messages sent to the chip is small, then there is no need for a clone manufacturer to crack the key. Instead, the clone manufacturer could incorporate a ROM in their chip that had a record of all of the responses from a genuine chip to the codes sent by the system. The larger the key, and the larger the response, the more space is required for such a lookup table.

5.7.1.12 Substitution with a sparse lookup table

If the messages sent to the chip are somehow predictable, rather than effectively random, then the clone manufacturer need not provide a complete lookup table. For example:

- If the message is simply a serial number, the clone manufacturer need simply provide a lookup table that contains values for past and predicted future serial numbers. There are unlikely to be more than 10^9 of these.
- If the test code is simply the date, then the clone manufacturer can produce a lookup table using the date as the address.
- If the test code is a pseudo-random number using either the serial number or the date as a seed, then the clone manufacturer just needs to crack the pseudo-random number generator in the system. This is probably not difficult, as they have access to the object code of the system. The clone manufacturer would then produce a content addressable memory (or other sparse array lookup) using these codes to access stored authentication codes.

5.7.1.13 Differential cryptanalysis

Differential cryptanalysis describes an attack where pairs of input streams are generated with known differences, and the differences in the encoded streams are analyzed.

Existing differential attacks are heavily dependent on the structure of S boxes, as used in DES and other similar algorithms. Although other algorithms such as HMAC-SHA1 have no S boxes, an attacker can undertake a differential-like attack by undertaking statistical analysis of:

- Minimal-difference inputs, and their corresponding outputs
- Minimal-difference outputs, and their corresponding inputs

Most algorithms were strengthened against differential cryptanalysis once the process was described. This is covered in the specific sections devoted to each cryptographic algorithm. However some recent algorithms developed in secret have been broken because the developers had not considered certain styles of differential attacks [94] and did not subject their algorithms to public scrutiny.

5.7.1.14 Message substitution attacks

In certain protocols, a man-in-the-middle can substitute part or all of a message. This is where a real authentication chip is plugged into a reusable clone chip within the consumable. The clone chip intercepts all messages between the system and the authentication chip, and can perform a number of substitution attacks.

Consider a message containing a header followed by content. An attacker may not be able to generate a valid header, but may be able to substitute their own content, especially if the valid response is something along the lines of "Yes, I received your message". Even if the return message is "Yes, I received the following message ...", the attacker may be able to substitute the original message before sending the acknowledgment back to the original sender.

Message Authentication Codes were developed to combat message substitution attacks.

5.7.1.15 Reverse engineering the key generator

If a pseudo-random number generator is used to generate keys, there is the potential for a clone manufacture to obtain the generator program or to deduce the random seed used. This was the way in which the security layer of the Netscape browser program was initially broken [33].

5.7.1.16 Bypassing the authentication process

It may be that there are problems in the authentication protocols that can allow a bypass of the authentication process altogether. With these kinds of attacks the key is completely irrelevant, and the attacker has no need to recover it or deduce it.

Consider an example of a system that authenticates at power-up, but does not authenticate at any other time. A reusable consumable with a clone authentication chip may make use of a real authentication chip. The clone authentication chip uses the real chip for the authentication call, and then simulates the real authentication chip's state data after that.

Another example of bypassing authentication is if the system authenticates only after the consumable has been used. A clone authentication chip can accomplish a simple authentication bypass by simulating a loss of connection after the use of the consumable but before the authentication protocol has completed (or even started).

One infamous attack known as the "Kentucky Fried Chip" hack [2] involved replacing a microcontroller chip for a satellite TV system. When a subscriber stopped paying the subscription fee, the system would send out a "disable" message. However the new micro-controller would simply detect this message and not pass it on to the consumer's satellite TV system.

5.7.1.17 Garrotelbribe attack

If people know the key, there is the possibility that they could tell someone else. The telling may be due to coercion (bribe, garrote etc.), revenge (e.g. a disgruntled employee), or simply for principle. These attacks are usually cheaper and easier than other efforts at deducing the key. As an example, a number of people claiming to be involved with the development of the (now defunct) Divx standard for DVD claimed (before the standard was rejected by consumers) that they would like to help develop Divx specific cracking devices - out of principle.

5.7.2 Physical attacks

The following attacks assume implementation of an authentication mechanism in a silicon chip that the attacker has physical access to. The first attack, ***Reading ROM***, describes an attack when keys are stored in ROM, while the remaining attacks assume that a secret key is stored in Flash memory.

5.7.2.1 Reading ROM

If a key is stored in ROM it can be read directly. A ROM can thus be safely used to hold a public key (for use in asymmetric cryptography), but not to hold a private key. In symmetric cryptography, a ROM is completely insecure. Using a copyright text (such as a *haiku*) as the key is not sufficient, because we are assuming that the cloning of the chip is occurring in a country where intellectual property is not respected.

5.7.2.2 Reverse engineering of chip

Reverse engineering of the chip is where an attacker opens the chip and analyzes the circuitry. Once the circuitry has been analyzed the inner workings of the chip's algorithm can be recovered.

Lucent Technologies have developed an active method [4] known as TOBIC (Two photon OBIC, where OBIC stands for Optical Beam Induced Current), to image circuits. Developed primarily for static RAM analysis, the process involves removing any back materials, polishing the back surface to a mirror finish, and then focusing light on the surface. The excitation wavelength is specifically chosen not to induce a current in the IC.

A Kerckhoffs in the nineteenth century made a fundamental assumption about cryptanalysis: *if the algorithm's inner workings are the sole secret of the scheme, the scheme is as good as broken* [39]. He stipulated that the secrecy must reside entirely in the key. As a result, the best way to protect against reverse engineering of the chip is to make the inner workings irrelevant.

5.7.2.3 Usurping the authentication process

It must be assumed that any clone manufacturer has access to both the system and consumable designs.

If the same channel is used for communication between the system and a trusted system authentication chip, and a non-trusted consumable authentication chip, it may be possible for the non-trusted chip to interrogate a trusted authentication chip in order to obtain the "correct answer". If this is so, a clone manufacturer would not have to determine the key. They would only have to trick the system into using the responses from the system authentication chip.

The alternative method of usurping the authentication process follows the same method as the logical attack described in Section 5.7.1.16 on page 23, involving simulated loss of contact with the system whenever authentication processes take place, simulating power-down etc.

5.7.2.4 Modification of system

This kind of attack is where the system itself is modified to accept clone consumables. The attack may be a change of system ROM, a rewiring of the consumable, or, taken to the extreme case, a completely clone system.

Note that this kind of attack requires each individual system to be modified, and would most likely require the owner's consent. There would usually have to be a clear advantage for the consumer to undertake such a modification, since it would typically void warranty and would most likely be costly. An example of such a modification with a clear advantage to the consumer is a software patch to change fixed-region DVD players into region-free DVD players (although it should be noted that this is not to use clone consumables, but rather originals from the same companies simply targeted for sale in other countries).

5.7.2.5 Direct viewing of chip operation by conventional probing

If chip operation could be directly viewed using an STM (Scanning Tunnelling Microscope) or an electron beam, the keys could be recorded as they are read from the internal non-volatile memory and loaded into work registers.

These forms of conventional probing require direct access to the top or front sides of the IC while it is powered.

5.7.2.6 Direct viewing of the non-volatile memory

If the chip were sliced so that the floating gates of the Flash memory were exposed, without discharging them, then the key could probably be viewed directly using an STM or SKM (Scanning Kelvin Microscope).

However, slicing the chip to this level without discharging the gates is probably impossible. Using wet etching, plasma etching, ion milling (focused ion beam etching), or chemical mechanical polishing will almost certainly discharge the small charges present on the floating gates.

5.7.2.7 Viewing the light bursts caused by state changes

Whenever a gate changes state, a small amount of infrared energy is emitted. Since silicon is transparent to infrared, these changes can be observed by looking at the circuitry from the underside of a chip. While the emission process is weak, it is bright enough to be detected by highly sensitive equipment developed for use in astronomy. The technique [92], developed by IBM, is called PICA (Picosecond Imaging Circuit Analyzer). If the state of a register is known at time t, then watching that register change over time will reveal the exact value at time $t+n$, and if the data is part of the key, then that part is compromised.

5.7.2.8 Viewing the keys using an SEPM

A non-invasive testing device, known as a Scanning Electric Potential Microscope (SEPM), allows the direct viewing of charges within a chip [37]. The SEPM has a tungsten probe that is placed a few micrometers above the chip, with the probe and circuit forming a capacitor. Any AC signal flowing beneath the probe causes displacement current to flow through this capacitor. Since the value of the current change depends on the amplitude and phase of the AC signal, the signal can be imaged. If the signal is part of the key, then that part is compromised.

5.7.2.9 Monitoring EMI

Whenever electronic circuitry operates, faint electromagnetic signals are given off. Relatively inexpensive equipment can monitor these signals and could give enough information to allow an attacker to deduce the keys.

5.7.2.10 Viewing I_{dd} fluctuations

Even if keys cannot be viewed, there is a fluctuation in current whenever registers change state. If there is a high enough signal to noise ratio, an attacker can monitor the difference in I_{dd} that may occur when programming over either a high or a low bit. The change in I_{dd} can reveal information about the key. Attacks such as these have already been used to break smart cards [46].

5.7.2.11 Differential Fault Analysis

This attack assumes introduction of a bit error by ionization, microwave radiation, or environmental stress. In most cases such an error is more likely to adversely affect the chip (e.g. cause the program code to crash) rather than cause beneficial changes which would

reveal the key. Targeted faults such as ROM overwrite, gate destruction etc. are far more likely to produce useful results.

5.7.2.12 Clock glitch attacks

Chips are typically designed to properly operate within a certain clock speed range. Some attackers attempt to introduce faults in logic by running the chip at extremely high clock speeds or introduce a clock glitch at a particular time for a particular duration [1]. The idea is to create race conditions where the circuitry does not function properly. An example could be an AND gate that (because of race conditions) gates through $Input_1$ all the time instead of the AND of $Input_1$ and $Input_2$.

If an attacker knows the internal structure of the chip, they can attempt to introduce race conditions at the correct moment in the algorithm execution, thereby revealing information about the key (or in the worst case, the key itself).

5.7.2.13 Power supply attacks

Instead of creating a glitch in the clock signal, attackers can also produce glitches in the power supply where the power is increased or decreased to be outside the working operating voltage range. The net effect is the same as a clock glitch - introduction of error in the execution of a particular instruction. The idea is to stop the CPU from XORing the key, or from shifting the data one bit-position etc. Specific instructions are targeted so that information about the key is revealed.

5.7.2.14 Overwriting ROM

Single bits in a ROM can be overwritten using a laser cutter microscope [1], to either 1 or 0 depending on the sense of the logic. If the ROM contains instructions, it may be a simple matter for an attacker to change a conditional jump to a non-conditional jump, or perhaps change the destination of a register transfer. If the target instruction is chosen carefully, it may result in the key being revealed.

5.7.2.15 Modifying EEPROM/Flash

These attacks fall into two categories:

- those similar to the ROM attacks except that the laser cutter microscope technique can be used to both set *and* reset individual bits. This gives much greater scope in terms of modification of algorithms.
- Electron beam programming of floating gates. As described in [89] and [32], a focused electron beam can change a gate by depositing electrons onto it. Damage to the rest of the circuit can be avoided, as described in [31].

5.7.2.16 Gate destruction

Anderson and Kuhn described the rump session of the 1997 workshop on Fast Software Encryption [1], where Biham and Shamir presented an attack on DES. The attack was to use a laser cutter to destroy an individual gate in the hardware implementation of a known block cipher (DES). The net effect of the attack was to force a particular bit of a register to be "stuck". Biham and Shamir described the effect of forcing a particular register to be affected in this way - the least significant bit of the output from the round function is set to 0. Comparing the 6 least significant bits of the left half and the right half can recover several bits of the key. Damaging a number of chips in this way can reveal enough information about the key to make complete key recovery easy.

An encryption chip modified in this way will have the property that encryption and decryption will no longer be inverses.

5.7.2.17 Overwrite attacks

Instead of trying to read the Flash memory, an attacker may simply set a single bit by use of a laser cutter microscope. Although the attacker doesn't know the previous value, they know the new value. If the chip still works, the bit's original state must be the same as the new state. If the chip doesn't work any longer, the bit's original state must be the logical NOT of the current state. An attacker can perform this attack on each bit of the key and obtain the n-bit key using at most *n* chips (if the new bit matched the old bit, a new chip is not required for determining the next bit).

5.7.2.18 Test circuitry attack

Most chips contain test circuitry specifically designed to check for manufacturing defects. This includes BIST (Built In Self Test) and scan paths. Quite often the scan paths and test circuitry includes access and readout mechanisms for all the embedded latches. In some cases the test circuitry could potentially be used to give information about the contents of particular registers.

Test circuitry is often disabled once the chip has passed all manufacturing tests, in some cases by blowing a specific connection within the chip. A determined attacker, however, can reconnect the test circuitry and hence enable it.

5.7.2.19 Memory remanence

Values remain in RAM long after the power has been removed [35], although they do not remain long enough to be considered non-volatile. An attacker can remove power once sensitive information has been moved into RAM (for example working registers), and then attempt to read the value from RAM. This attack is most useful against security systems that have regular RAM chips. A classic example is cited by [1], where a security system was designed with an automatic power-shut-off that is triggered when the computer case is opened. The attacker was able to simply open the case, remove the RAM chips, and retrieve the key because the values persisted.

5.7.2.20 Chip theft attack

If there are a number of stages in the lifetime of an authentication chip, each of these stages must be examined in terms of ramifications for security should chips be stolen. For example, if information is programmed into the chip in stages, theft of a chip between stages may allow an attacker to have access to key information or reduced efforts for attack. Similarly, if a chip is stolen directly after manufacture but before programming, does it give an attacker any logical or physical advantage?

5.7.2.21 Trojan horse attack

At some stage the authentication chips must be programmed with a secret key. Suppose an attacker builds a clone authentication chip and adds it to the pile of chips to be programmed. The attacker has especially built the clone chip so that it looks and behaves just like a real authentication chip, but will give the key out to the attacker when a special attacker-known command is issued to the chip. Of course the attacker must have access to the chip after the programming has taken place, as well as physical access to add the Trojan horse authentication chip to the genuine chips.

6 Requirements

Existing solutions to the problem of authenticating consumables have typically relied on patents covering physical packaging. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. Consequently a much higher level of protection is required.

The authentication mechanism is therefore built into an authentication chip that is embedded in the consumable and allows a system to authenticate that consumable securely and easily. Limiting ourselves to the system authenticating consumables (we don't consider the consumable authenticating the system), two levels of protection can be considered:

Presence Only Authentication:

This is where only the presence of an authentication chip is tested. The authentication chip can be removed and used in other consumables as long as be used indefinitely.

Consumable Lifetime Authentication:

This is where not only is the presence of the authentication chip tested for, but also the authentication chip must only last the lifetime of the consumable. For the chip to be re-used it must be completely erased and reprogrammed.

The two levels of protection address different requirements. We are primarily concerned with Consumable Lifetime authentication in order to prevent cloned versions of high volume consumables. In this case, each chip should hold secure state information about the consumable being authenticated. It should be noted that a Consumable Lifetime authentication chip could be used in any situation requiring a Presence Only authentication chip.

Requirements for authentication, data storage integrity and manufacture are considered separately. The following sections summarize requirements of each.

6.1 AUTHENTICATION

The authentication requirements for both Presence Only and Consumable Lifetime authentication are restricted to the case of a system authenticating a consumable. We do not consider bi-directional authentication where the consumable also authenticates the system. For example, it is not necessary for a valid toner cartridge to ensure it is being used in a valid photocopier.

For Presence Only authentication, we must be assured that an authentication chip is physically present. For Consumable Lifetime authentication we also need to be assured that state data actually came from the authentication chip, and that it has not been altered en route. These issues cannot be separated - data that has been altered has a new source, and if the source cannot be determined, the question of alteration cannot be settled.

It is not enough to provide an authentication method that is secret, relying on a home-brew security method that has not been scrutinized by security experts. The primary requirement therefore is to provide authentication by means that have withstood the scrutiny of experts.

The authentication scheme used by the authentication chip should be resistant to defeat by logical means. Logical types of attack are extensive, and attempt to do one of three things:

- Bypass the authentication process altogether

- Obtain the secret key by force or deduction, so that any question can be answered
- Find enough about the nature of the authenticating questions and answers in order to, without the key, give the right answer to each question.

The logical attack styles and the forms they take are detailed in Section 5.7.1 on page 19.

The algorithm should have a flat keyspace, allowing any random bit string of the required length to be a possible key. There should be no weak keys.

6.2 DATA STORAGE INTEGRITY

Although authentication protocols take care of ensuring data integrity in communicated messages, data storage integrity is also required. Two kinds of data must be stored within the authentication chip:

- Authentication data, such as secret keys
- Consumable state data, such as serial numbers, and media remaining etc.

The access requirements of these two data types differ greatly. The authentication chip therefore requires a storage/access control mechanism that allows for the integrity requirements of each type.

6.2.1 Authentication data

Authentication data must remain confidential. It needs to be stored in the chip during a manufacturing/programming stage of the chip's life, but from then on must not be permitted to leave the chip. It must be resistant to being read from non-volatile memory. The authentication scheme is responsible for ensuring the key cannot be obtained by deduction, and the manufacturing process is responsible for ensuring that the key cannot be obtained by physical means.

The size of the authentication data memory area must be large enough to hold the necessary keys and secret information as mandated by the authentication protocols.

6.2.2 Consumable state data

Consumable state data can be divided into the following types. Depending on the application, there will be different numbers of each of these types of data items.

- Read Only
- ReadWrite
- Decrement Only

Read Only data needs to be stored in the chip during a manufacturing/programming stage of the chip's life, but from then on should not be allowed to change. Examples of Read Only data items are consumable batch numbers and serial numbers.

ReadWrite data is changeable state information, for example, the last time the particular consumable was used. ReadWrite data items can be read and written an unlimited number of times during the lifetime of the consumable. They can be used to store any state information about the consumable. The only requirement for this data is that it needs to be kept in non-volatile memory. Since an attacker can obtain access to a system (which can write to ReadWrite data), any attacker can potentially change data fields of this type. This data type should not be used for secret information, and must be considered insecure.

Decrement Only data is used to count down the availability of consumable resources. A photocopier's toner cartridge, for example, may store the amount of toner remaining as a Decrement Only data item. An ink cartridge for a color printer may store the amount of each ink color as a Decrement Only data item, requiring 3 (one for each of Cyan, Magenta, and Yellow), or even as many as 5 or 6 Decrement Only data items. The requirement for this kind of data item is that once programmed with an initial value at the manufacturing/programming stage, *it can only reduce in value*. Once it reaches the minimum value, it cannot decrement any further. The Decrement Only data item is only required by Consumable Lifetime authentication.

Note that the size of the consumable state data storage required is only for that information required to be authenticated. Information which would be of no use to an attacker, such as ink color-curve characteristics or ink viscosity do not have to be stored in the secure state data memory area of the authentication chip.

6.3 MANUFACTURE

The authentication chip must have a low manufacturing cost in order to be included as the authentication mechanism for low cost consumables.

The authentication chip should use a standard manufacturing process, such as Flash. This is necessary to:

- Allow a great range of manufacturing location options
- Use well-defined and well-behaved technology
- Reduce cost

Regardless of the authentication scheme used, the circuitry of the authentication part of the chip must be resistant to physical attack. Physical attack comes in four main ways, although the form of the attack can vary:

- Bypassing the authentication chip altogether
- Physical examination of chip while in operation (destructive and non-destructive)
- Physical decomposition of chip
- Physical alteration of chip

The physical attack styles and the forms they take are detailed in Section 5.7.2 on page 23.

Ideally, the chip should be exportable from the USA, so it should not be possible to use an authentication chip as a secure encryption device. This is low priority requirement since there are many companies in other countries able to manufacture the authentication chips. In any case, the export restrictions from the USA may change.

AUTHENTICATION

7 Introduction

Existing solutions to the problem of authenticating consumables have typically relied on physical patents on packaging. However this does not stop home refill operations or clone manufacture in countries with weak industrial property protection. Consequently a much higher level of protection is required.

It is not enough to provide an authentication method that is secret, relying on a home-brew security method that has not been scrutinized by security experts. Security systems such as Netscape's original proprietary system and the GSM Fraud Prevention Network used by cellular phones are examples where design secrecy caused the vulnerability of the security [33][94]. Both security systems were broken by conventional means that would have been detected if the companies had followed an open design process. The solution is to provide authentication by means that have withstood the scrutiny of experts.

> ***It is possible to build a cabin with no foundations, but not a lasting building.***
> - Eng. Isidor Goldreich (1906-1995)

In this section, we examine a number of protocols that can be used for consumables authentication. We only use security methods that are publicly described, using known behaviors in this new way. Readers should be familiar with the concepts and terms described in Section 5 on page 5. We avoid the Zero Knowledge Proof protocol since it is patented.

For all protocols, the security of the scheme relies on a secret key, not a secret algorithm. In the nineteenth century, A Kerckhoffs made a fundamental assumption about cryptanalysis: *if the algorithm's inner workings are the sole secret of the scheme, the scheme is as good as broken* [39]. He stipulated that the secrecy must reside entirely in the key. As a result, the best way to protect against reverse engineering of any authentication chip is to make the algorithmic inner workings irrelevant (the algorithm of the inner workings must still be must be valid, but not the actual secret).

The QA Chip is a programmable device, and can therefore be setup with an application-specific program together with an application-specific set of protocols. This section describes the following sets of protocols:

- single key single memory vector
- multiple key single memory vector
- multiple key multiple memory vector

These protocols refer to the number of valid keys that an QA Chip knows about, and the size of data required to be stored in the chip.

From these protocols it is straightforward to construct protocol sets for the single key multiple memory vector case (of course the multiple memory vector can be considered to be . and multiple key single memory vector. Other protocol sets can also be defined as necessary. Of course multiple memory vector can be conveniently

All the protocols rely on a time-variant challenge (i.e. the challenge is different each time), where the response depends on the challenge and the secret. The challenge involves a random number so that any observer will not be able to gather useful information about a subsequent identification.

8 Single Key Single Memory Vector

8.1 PROTOCOL BACKGROUND

This protocol set is provided for two reasons:

- the other protocol sets defined in this document are simply extensions of this one; and
- it is useful in its own right

The single key protocol set is useful for applications where only a single key is required. Note that there can be many consumables and systems, but there is only a single key that connects them all. Examples include:

- car and keys. A car and the car-key share a single key. There can be multiple sets of car-keys, each effectively cut to the same key. A company could have a set of cars, each with the same key. Any of the car-keys could then be used to drive any of the cars.
- printer and ink cartridge. All printers of a certain model use the same ink cartridge, with printer and cartridge sharing only a single key. Note that to introduce a new printer model that accepts the old ink cartridge the new model would need the same key as the old model. See the multiple-key protocols for alternative solutions to this problem.

8.2 REQUIREMENTS OF PROTOCOL

Each QA Chip contains the following values:

K	The secret key for calculating $F_K[X]$. K must not be stored directly in the QA Chip. Instead, each chip needs to store a random number R_K (different for each chip), $K \oplus R_K$, and $\neg K \oplus R_K$. The stored $K \oplus R_K$ can be XORed with R_K to obtain the real K. Although $\neg K \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M	Memory vector of QA Chip.
P	2 element array of access permissions for each part of M. Entry 0 holds access permissions for non-authenticated writes to M (no key required). Entry 1 holds access permissions for authenticated writes to M (key required). Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.
C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a submessage to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_K[X]$	*Internal function only.* Returns $S_K[X]$, the result of applying a digital signature function S to X based upon key K. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1 (see Section 13 on page 61), and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

8.3 READS OF M

In this case, we have a trusted chip (*ChipT*) connected to a System. The System wants to authenticate an object that contains a non-trusted chip (*ChipA*). In effect, the System wants to know that it can securely read a memory vector (M) from ChipA: to be sure that ChipA is valid and that M has not been altered.

The protocol requires the following publicly available function in ChipA:

Read[X] Advances R, and returns R, M, $S_K[X|R|C_1|M]$. The time taken to calculate the signature must not be based on the contents of X, R, M, or K.

The protocol requires the following publicly available functions in ChipT:

Random[] Returns R (does not advance R).

Test[X, Y, Z] Advances R and returns 1 if $S_K[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

To authenticate ChipA and read ChipA's memory M:

a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipA's Read function, passing in the result from b;

d. ChipA updates R_A, then calculates and returns R_A, M_A, $S_K[R_T|R_A|C_1|M_A]$;

e. System calls ChipT's Test function, passing in R_A, M_A, $S_K[R_T|R_A|C_1|M_A]$;

f. System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

The data flow for read authentication is shown in Figure 1:

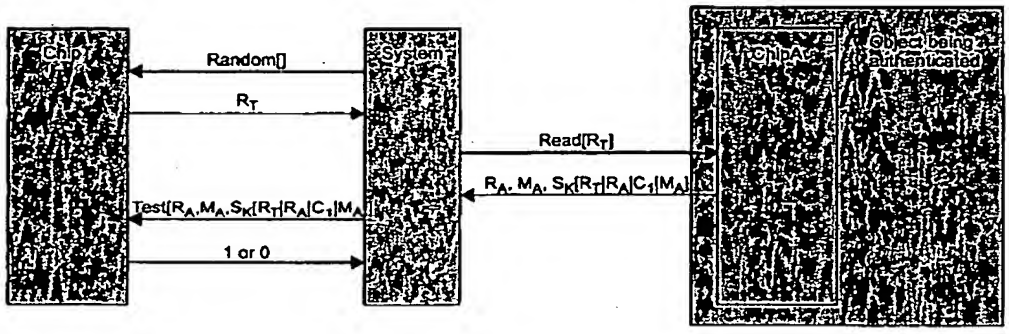


Figure 1. Protocol for single key authenticated read

The protocol allows System to simply pass data from one chip to another, with no special processing. The protection relies on ChipT being trusted, even though System does not know K.

When ChipT is physically separate from System (eg is chip on a board connected to System) System ***must also occassionally*** (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to prevent someone from inserting a fake ChipT into the system that always returns 1 for the Test function.

8.4 WRITES

In this case, the System wants to update M in some chip referred to as *ChipU*. This can be non-authenticated (for example, anyone is allowed to count down the amount of consumable remaining), or authenticated (for example, replenishing the amount of consumable remaining).

8.4.1 Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation. In this kind of write, System wants to change M in a way that doesn't require special authorization. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know K or even have access to a trusted chip*, System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires the following publicly available function:

Write[X] Writes X over those parts of M subject to P_0 and the existing value for M.

To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};

b. The authentication procedure for a Read is carried out (see Section 8.3 on page 34);

c. If ChipU is authentic and M_{new} = M returned in **b**, the write succeeded. If not, it failed.

8.4.2 Authenticated writes

In this kind of write, System wants to change Chip U's M in an authorized way, without being subject to the permissions that apply during normal operation (P_0). For example, the consumable may be at a refilling station and the normally Decrement Only section of M should be updated to include the new valid consumable. In this case, the chip whose M is being updated must authenticate the writes being generated by the external System and in addition, apply permissions P_1 to ensure that only the correct parts of M are updated.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Read[X] Advances R, and returns R, M, $S_K[X|R|C_1|M]$. The time taken to calculate the signature must be identical for all inputs.

WriteA[X, Y, Z] Returns 1, advances R, and replaces M by Y subject to P_1 only if $S_K[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y over those parts of M subject to P_1 when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once

ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P_1 allows that part of M to be updated).

Q Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), followed by $S_K[W|R|C_1|Z_{QX}]$ only if $S_K[V|W|C_1|X] = Y$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in 0 as the input parameter;

b. ChipU produces R_U, M_U, $S_K[0|R_U|C_1|M_U]$ and returns these to System;

c. System calls ChipS's SignM function, passing in 0 (as used in a), R_U, M_U, $S_K[0|R_U|C_1|M_U]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_U using Q) and $S_K[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in d are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The data flow for authenticated writes is shown in Figure 2:

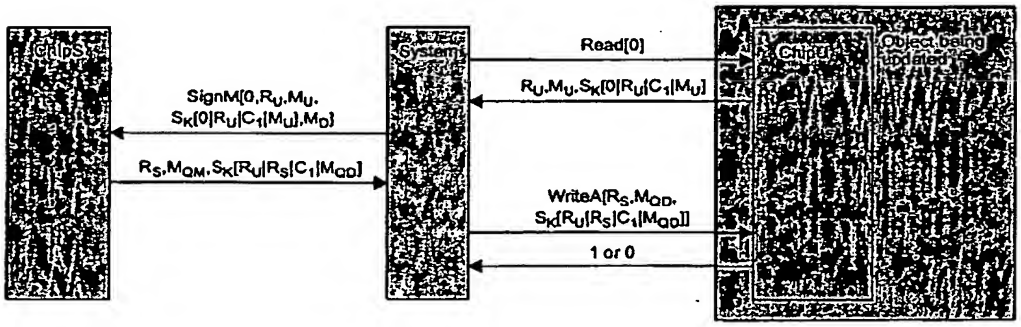


Figure 2. Protocol for single key authenticated write

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by P_0 set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

The same is true of CountRemaining. The CountRemaining value needs to be setup (including making it ReadOnly in P_0) before ChipS is programmed with K_U. ChipS is therefore programmed to only perform a limited number of SignM operations (thereby

limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

8.4.3 Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P and certain parts of M, only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all parts of M. As different processes fill up different parts of M, they can be sealed against future change by updating the permissions. Updating a chip's P_0 changes permissions for unauthorized writes, and updating P_1 changes permissions for authorized writes.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Random[] Returns R (does not advance R).

SetPermission[n,X,Y,Z] Advances R, and updates P_n according to Y and returns 1 followed by the resultant P_n only if $S_K[R|X|Y|C_2] = Z$. Otherwise returns 0. P_n can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_n).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P_1 allows that part of M to be updated).

SignP[X,Y] Advances R, decrements CountRemaining and returns R and $S_K[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's P_n:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_K[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with the desired n, R_S, P_D and $S_K[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_K[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The data flow for authenticated writes to permissions is shown in Figure 3:

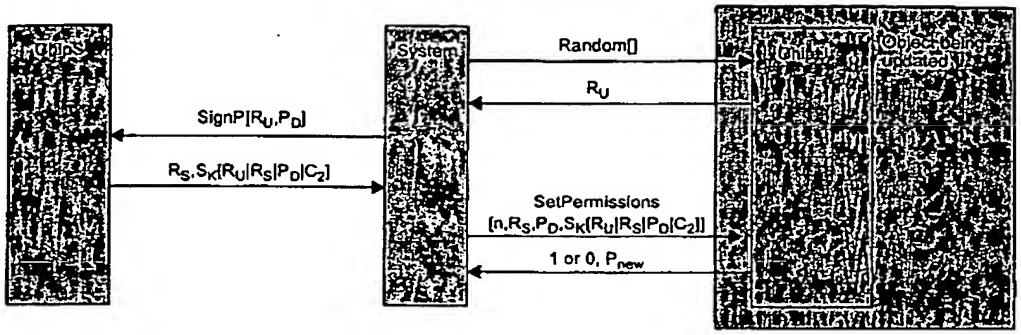


Figure 3. Protocol for single key update of permissions

8.5 PROGRAMMING K

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

The protocol assumes that ChipF and ChipP already share a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

The protocol requires the following publicly available functions in ChipP:

Random[] Returns R (does not advance R).

ReplaceKey[X, Y, Z] Replaces K by $S_{Kold}[R|X|C_3]\oplus Y$, advances R, and returns 1 only if $S_{Kold}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

And the following data and function in ChipF:

CountRemaining Part of M with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M.

K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible.

SetPartialKey[X,Y] If word X of K_{new} has not yet been set, set word X of K_{new} to Y and return 1. Otherwise return 0. This function allows K_{new} to be programmed in multiple steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory. Since there is a small number of ChipFs, it is theoretically not necessary to store the inverse of K_{new}, but it is stronger protection to do so.

GetProgramKey[X] Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kold}[X|R_F|C_3]\oplus K_{new}$ and a signature of the first two

outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

To update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in the result from **b**;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kold}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{Kold}[R_F|S_{Kold}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in the response from **d**;

f. System checks response from ChipP. If the response is 1, then K_P has been correctly updated to K_{new}. If the response is 0, K_P has not been updated.

The data flow for key updates is shown in Figure 4:

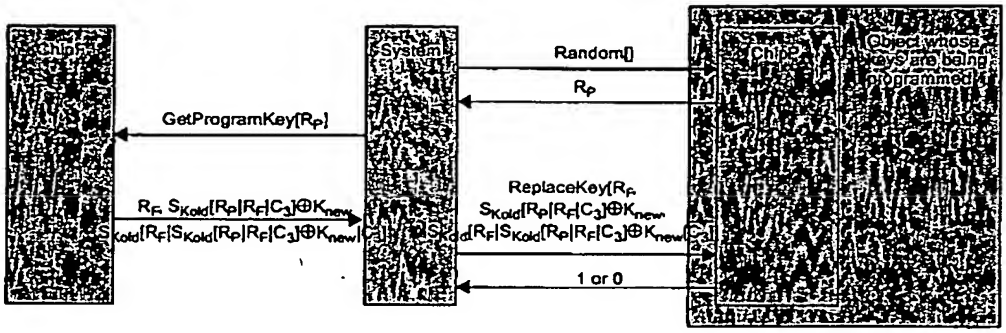


Figure 4. Protocol for single key update

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kold}[R_P|R_F|C_3]$ without K_{old}. The third parameter, a signature, is sent to ensure that ChipP can determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_F (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if neccesary (see Section 8.4 on page 35).

8.5.1 Chicken and Egg

Of course, for the Program Key protocol to work, both ChipF and ChipP must both know K_{old}. Obviously both chips had to be programmed with K_{old}, and thus K_{old} can be thought of as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc, and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks.

9 Multiple Key Single Memory Vector

9.1 PROTOCOL BACKGROUND

This protocol set is an extension to the single key single memory vector protocol set, and is provided for two reasons:

- the multiple key multiple memory vector protocol set defined in this document is simply extensions of this one; and
- it is useful in its own right

The multiple key protocol set is typically useful for applications where there are multiple types of systems and consumables, and they need to work with each other in various ways. This is typically in the following situations:

- when different systems want to share some consumables, but not others. For example printer models may share some ink cartridges and not share others.
- when there are different owners of data in M. Part of the memory vector may be owned by one company (eg the speed of the printer) and another may be owned by another (eg the serial number of the chip). In this case a given key K_n needs to be able to write to a given part of M, and other keys K_n need to be disallowed from writing to these same areas.

9.2 REQUIREMENTS OF PROTOCOL

Each QA Chip contains the following values:

N	The maximum number of keys known to the chip.
K_N	Array of *N* secret keys used for calculating $F_{Kn}[X]$ where K_n is the *n*th element of the array. Each K_n must not be stored directly in the QA Chip. Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M	Memory vector of QA Chip. A fixed part of M contains N in ReadOnly form so users of the chip can know the number of keys known by the chip.
P	N+1 element array of access permissions for each part of M. Entry 0 holds access permissions for non-authenticated writes to M (no key required). Entries 1 to N+1 hold access permissions for authenticated writes to M, one for each K. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.
C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a submessage to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_{Kn}[N,X]$	*Internal function only.* Returns $S_{Kn}[X]$, the result of applying a digital signature function S to X based upon the appropriate key K_n. The digital

signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1 (see Section 13 on page 61), and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

9.3 READS

As with the single key scenario, we have a trusted chip (*ChipT*) connected to a System. The System wants to authenticate an object that contains a non-trusted chip (*ChipA*). In effect, the System wants to know that it can securely read a memory vector (M) from ChipA: to be sure that ChipA is valid and that M has not been altered.

The protocol requires the following publicly available functions:

Random[] Returns R (does not advance R).

Read[n, X] Advances R, and returns R, M, $S_{Kn}[X|R|C_1|M]$. The time taken to calculate the signature must not be based on the contents of X, R, M, or K.

Test[n,X, Y, Z] Advances R and returns 1 if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

To authenticate ChipA and read ChipA's memory M:

a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipA's Read function, passing in some key number n1 and the result from b;

d. ChipA updates R_A, then calculates and returns R_A, M_A, $S_{KAn1}[R_T|R_A|C_1|M_A]$;

e. System calls ChipT's Test function, passing in n2, R_A, M_A, $S_{KAn1}[R_T|R_A|C_1|M_A]$;

f. System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

The choice of n1 and n2 must be such that ChipA's K_{n1} = ChipT's K_{n2}.

The data flow for read authentication is shown in Figure 5:

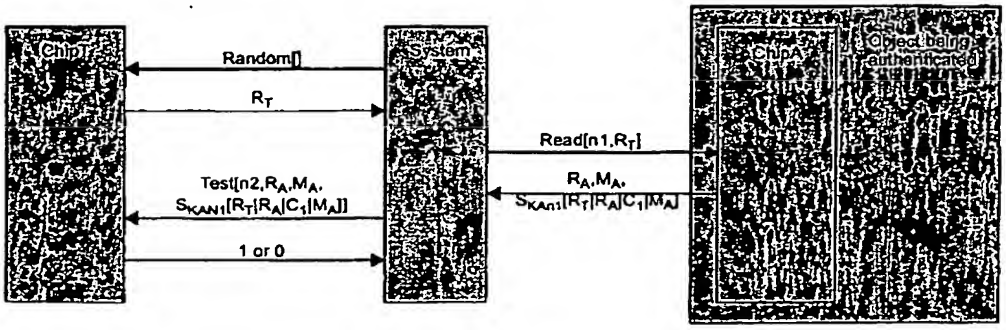


Figure 5. Protocol for multiple key single M authenticated read

The protocol allows System to simply pass data from one chip to another, with no special processing. The protection relies on ChipT being trusted, even though System does not know K.

When ChipT is physically separate from System (eg is chip on a board connected to System) System ***must also occassionally*** (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to prevent someone from inserting a fake ChipT into the system that always returns 1 for the Test function.

It is important that n1 is chosen by System. Otherwise ChipA would need to return N_A sets of signatures for each read, since ChipA does not know which of the keys will satisfy ChipT. Similarly, system must also choose n2, so it can potentially restrict the number of keys in ChipT that are matched against (otherwise ChipT would have to match against all its keys). This is important in order to restrict how different keys are used. For example, say that ChipT contains 6 keys, keys 0-2 are for various printer-related upgrades, and keys 3-6 are for inks. ChipA contains say 4 keys, one key for each printer model. At power-up, System goes through each of chipA's keys 0-3, trying each out against ChipT's keys 3-6. System doesn't try to match against ChipT's keys 0-2. Otherwise knowledge of a speed-upgrade key could be used to provide ink QA Chip chips. This matching needs to be done only once (eg at power up). Once matching keys are found, System can continue to use those key numbers.

Since System needs to know N_T and N_A, part of M is used to hold N (eg in Read Only form), and the system can obtain it by calling the Read function, passing in key 0.

9.4 WRITES

As with the single key scenario, the System wants to update M in ChipU. As before, this can be done in a non-authenticated and authenticated way.

9.4.1 Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation. In this kind of write, System wants to change M subject to P. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know any of the Ks or even have access to a trusted chip* to perform the write, System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires the following publicly available function:

Write[X] Writes X over those parts of M subject to P_0 and the existing value for M.

To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};

b. The authentication procedure for a Read is carried out (see Section 9.3 on page 41);

c. If ChipU is authentic and M_{new} = M returned in b, the write succeeded. If not, it failed.

9.4.2 Authenticated writes

In this kind of write, System wants to change Chip U's M in an authorized way, without being subject to the permissions that apply during normal operation (P_0). For example, the consumable may be at a refilling station and the normally Decrement Only section of M should be updated to include the new valid consumable. In this case, the chip whose M is being updated must authenticate the writes being generated by the external System and in addition, apply the appropriate permission for the key to ensure that only the correct parts of M are updated. Having a different permission for each key is required as when multiple keys are involved, all keys should not necessarily be given open access to M. For example, suppose M contains printer speed and a counter of money available for franking. A ChipS that updates printer speed should not be capable of updating the amount of money. Since P_0 is used for non-authenticated writes, each K_n has a corresponding permission P_{n+1} that determines what can be updated in an authenticated write.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Read[n, X] Advances R, and returns R, M, $S_{Kn}[X|R|C_1|M]$. The time taken to calculate the signature must not be based on the contents of X, R, M, or K.

WriteA[n, X, Y, Z] Advances R, replaces M by Y subject to P_{n+1}, and returns 1 only if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y subject to P_{n+1} to its M when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P allows that part of M to be updated).

Q Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[n,V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{Kn}[W|R|C_1|Z_{QX}]$ only if Y= $S_{Kn}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1 and 0 as the input parameters;

b. ChipU produces R_U, M_U, $S_{Kn1}[0|R_U|C_1|M_U]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (as used in a), R_U, M_U, $S_{Kn1}[0|R_U|C_1|M_U]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_U using Q) and $S_{Kn2}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in **d** are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from **d**.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes is shown in Figure 6:

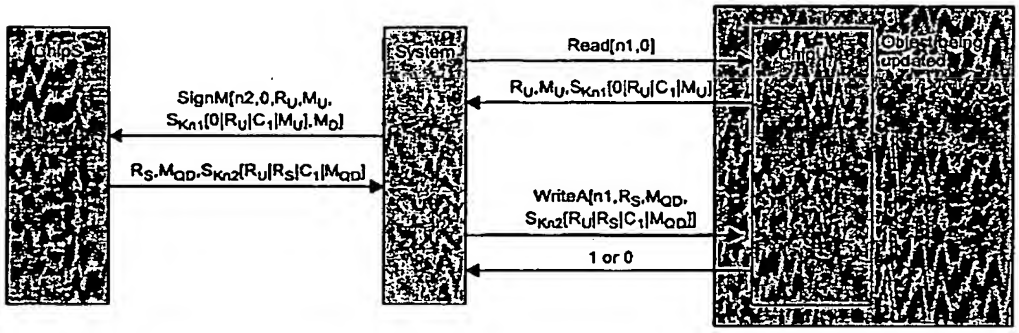


Figure 6. Protocol for multiple key authenticated write

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by P_0 set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

In addition, CountRemaining in ChipS needs to be setup (including making it ReadOnly in P_S) before ChipS is programmed with K_U. ChipS should therefore be programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

9.4.3 Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P (and certain parts of M), only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all parts of M. As different processes fill up different parts of M, they can be sealed against future change by updating the permissions. Updating a chip's P_0 changes permissions for unauthorized writes, and updating P_{n+1} changes permissions for authorized writes with key K_n.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Random[] Returns R (does not advance R).

SetPermission[n,p,X,Y,Z] Advances R, and updates P_p according to Y and returns 1 followed by the resultant P_p only if $S_{Kn}[R|X|Y|C_2] = Z$. Otherwise returns 0. P_p can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_p).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's P_0 for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M (assuming ChipS's P_n allows that part of M to be updated).

SignP[n,X,Y] Advances R, decrements CountRemaining and returns R and $S_{Kn}[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's P_n:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in n1, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{Kn1}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in d are non zero, then System can then call ChipU's SetPermission function with n2, the desired permission entry p, R_S, P_D and $S_{Kn1}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_{Kn2}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes to permissions is shown in Figure 7:

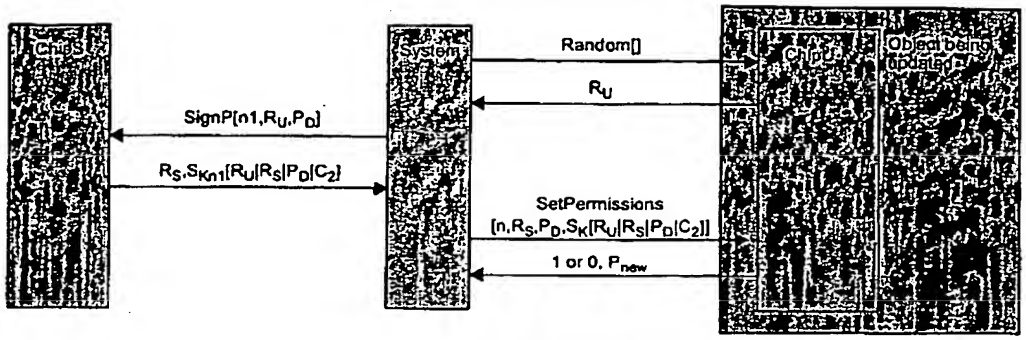


Figure 7. Protocol for multiple key update of permissions

9.4.4 Protecting M in a multiple key system

To protect the appropriate part of M, the SetPermission function must be called *after* the part of M has been set to the desired value.

For example, if adding a serial number to an area of M that is currently ReadWrite so that noone is permitted to update the number again:

- the Write function is called to write the serial number to M
- SetPermission is called for n = {1, ..., N} to set that part of M to be ReadOnly for authorized writes using key n-1.
- SetPermission is called for 0 to set that part of M to be ReadOnly for non-authorized writes

For example, adding a consumable value to M such that only keys 1-2 can update it, and keys 0, and 3-N cannot:

- the Write function is called to write the amount of consumable to M
- SetPermission is called for n = {1, 4, 5, ..., N-1} to set that part of M to be ReadOnly for authorized writes using key n-1. This leaves keys 1 and 2 with ReadWrite permissions.
- SetPermission is called for 0 to set that part of M to be DecrementOnly for non-authorized writes. This allows the amount of consumable to decrement.

It is possible for someone who knows a key to further restrict other keys, but it is not in anyone's interest to do so.

9.5 PROGRAMMING K

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

The protocol is a simple extension of the single key protocol in that it assumes that ChipF and ChipP already share a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

The protocol requires the following publicly available functions in ChipP:

Random[] Returns R (does not advance R).

ReplaceKey[n, X, Y, Z] Replaces K_n by $S_{Kn}[R|X|C_3]\oplus Y$, advances R, and returns 1 only if $S_{Kn}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

And the following data and functions in ChipF:

CountRemaining Part of M with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M.

K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible.

SetPartialKey[X,Y] If word X of K_{new} has not yet been set, set word X of K_{new} to Y and return 1. Otherwise return 0. This function allows K_{new} to be programmed in multiple steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory. Since there is a small number of ChipFs, it is theoretically not necessary to store the inverse of K_{new}, but it is stronger protection to do so.

GetProgramKey[n, X] Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kn}[X|R_F|C_3]\oplus K_{new}$ and a signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

To update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in n1 (the desired key to use) and the result from **b**;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{Kn1}[R_F|S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in n2 (the key to use in ChipP) and the response from **d**;

f. System checks response from ChipP. If the response is 1, then K_{Pn2} has been correctly updated to K_{new}. If the response is 0, K_{Pn2} has not been updated.

The choice of n1 and n2 must be such that ChipF's K_{n1} = ChipP's K_{n2}.

The data flow for key updates is shown in Figure 8:

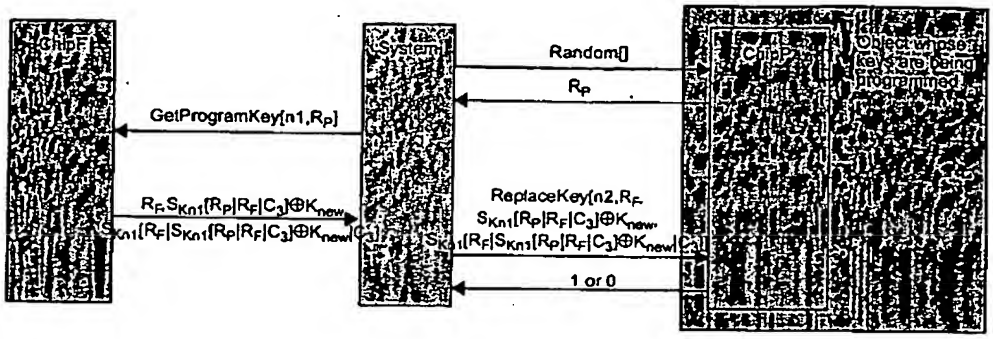


Figure 8. Protocol for multiple key update

Note that K_{new} is never passed in the open. An attacker could send its own R_P but cannot produce $S_{Kn1}[R_P|R_F|C_3]$ without K_{n1}. The signature based on K_{new} is sent to ensure that ChipP will be able to determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_F (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if neccesary (see Section 9.4 on page 42).

9.5.1 Chicken and Egg

As with the single key protocol, for the Program Key protocol to work, both ChipF and ChipP must both know K_{old}. Obviously both chips had to be programmed with K_{old}, and thus K_{old} can be thought of as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc, and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks.

Depending on the reprogramming requirements, K_{first} can be the same or different for all K_n.

10 Multiple Keys Multiple Memory Vectors

10.1 PROTOCOL BACKGROUND

This protocol set is a slight restriction of the multiple key single memory vector protocol set, and is the expected protocol. It is a restriction in that M has been optimized for Flash memory utilization.

M is broken into multiple memory vectors (semi-fixed and variable components) for the purposes of optimizing flash memory utilization. Typically M contains some parts that are fixed at some stage of the manufacturing process (eg a batch number, serial number etc), and once set, are not ever updated. This information does not contain the amount of consumable remaining, and therefore is not read or written to with any great frequency.

We therefore define M_0 to be the M that contains the frequently updated sections, and the remaining Ms to be rarely written to. Authenticated writes only write to M_0, and non-authenticated writes can be directed to a specific M_n. This reduces the size of permissions that are stored in the QA Chip (since key-based writes are not required for Ms other than M_0). It also means that M_0 and the remaining Ms can be manipulated in different ways, thereby increasing flash memory longevity.

10.2 REQUIREMENTS OF PROTOCOL

Each QA Chip contains the following values:

N	The maximum number of keys known to the chip.
T	The number of vectors M is broken into.
K_N	Array of *N* secret keys used for calculating $F_{Kn}[X]$ where K_n is the *n*th element of the array. Each K_n must not be stored directly in the QA Chip. Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M_T	Array of T memory vectors. Only M_0 can be written to with an authorized write, while all Ms can be written to in an unauthorized write. Writes to M_0 are optimized for Flash usage, while updates to any other M_n are expensive with regards to Flash utilization, and are expected to be only performed once per section of M_n. M_1 contains T and N in ReadOnly form so users of the chip can know these two values.
P_{T+N}	T+N element array of access permissions for each part of M. Entries n={0... T-1} hold access permissions for non-authenticated writes to M_n (no key required). Entries n={T to T+N-1}hold access permissions for authenticated writes to M_0 for K_n. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only.
C	3 constants used for generating signatures. C_1, C_2, and C_3 are constants that pad out a submessage to a hashing boundary, and all 3 must be different.

Each QA Chip contains the following private function:

$S_{Kn}[N,X]$ *Internal function only.* Returns $S_{Kn}[X]$, the result of applying a digital signature function S to X based upon the appropriate key K_n. The digital signature must be long enough to counter the chances of someone generating a random signature. The length depends on the signature scheme chosen, although the scheme chosen for the QA Chip is HMAC-SHA1, and therefore the length of the signature is 160 bits.

Additional functions are required in certain QA Chips, but these are described as required.

10.3 READS

As with the previous scenarios, we have a trusted chip (*ChipT*) connected to a System. The System wants to authenticate an object that contains a non-trusted chip (*ChipA*). In effect, the System wants to know that it can securely read a memory vector (M_t) from ChipA: to be sure that ChipA is valid and that M has not been altered.

The protocol requires the following publicly available functions:

Random[] Returns R (does not advance R).

Read[n, t, X] Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K. If t is invalid, the function assumes t=0.

Test[n,X, Y, Z] Advances R and returns 1 if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content.

To authenticate ChipA and read ChipA's memory M:

a. System calls ChipT's Random function;

b. ChipT returns R_T to System;

c. System calls ChipA's Read function, passing in some key number n1, the desired M number t, and the result from b;

d. ChipA updates R_A, then calculates and returns R_A, M_{At}, $S_{KAn1}[R_T|R_A|C_1|M_{At}]$;

e. System calls ChipT's Test function, passing in n2, R_A, M_{At}, $S_{KAn1}[R_T|R_A|C_1|M_{At}]$;

f. System checks response from ChipT. If the response is 1, then ChipA is considered authentic. If 0, ChipA is considered invalid.

The choice of n1 and n2 must be such that ChipA's K_{n1} = ChipT's K_{n2}.

The data flow for read authentication is shown in Figure 5:

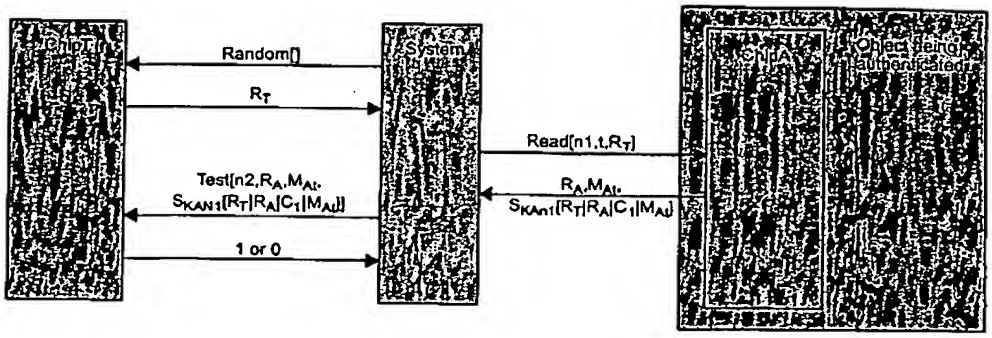


Figure 9. Protocol for multiple key muliple M authenticated read

The protocol allows System to simply pass data from one chip to another, with no special processing. The protection relies on ChipT being trusted, even though System does not know K.

When ChipT is physically separate from System (eg is chip on a board connected to System) System ***must also occassionally*** (based on system clock for example) call ChipT's Test function with bad data, expecting a 0 response. This is to prevent someone from inserting a fake ChipT into the system that always returns 1 for the Test function.

It is important that n1 is chosen by System. Otherwise ChipA would need to return N_A sets of signatures for each read, since ChipA does not know which of the keys will satisfy ChipT. Similarly, system must also choose n2, so it can potentially restrict the number of keys in ChipT that are matched against (otherwise ChipT would have to match against all its keys). This is important in order to restrict how different keys are used. For example, say that ChipT contains 6 keys, keys 0-2 are for various printer-related upgrades, and keys 3-6 are for inks. ChipA contains say 4 keys, one key for each printer model. At power-up, System goes through each of chipA's keys 0-3, trying each out against ChipT's keys 3-6. System doesn't try to match against ChipT's keys 0-2. Otherwise knowledge of a speed-upgrade key could be used to provide ink QA Chip chips. This matching needs to be done only once (eg at power up). Once matching keys are found, System can continue to use those key numbers.

Since System needs to know N_T, N_A, and T_A, part of M_1 is used to hold N (eg in Read Only form), and the system can obtain it by calling the Read function, passing in key 0 and t=1.

10.4 WRITES

As with the previous scenarios, the System wants to update M_t in ChipU. As before, this can be done in a non-authenticated and authenticated way.

10.4.1 Non-authenticated writes

This is the most frequent type of write, and takes place between the System / consumable during normal everyday operation for M_0, and during the manufacturing process for M_t.

In this kind of write, System wants to change M subject to P. For example, the System could be decrementing the amount of consumable remaining. Although *System does not need to know and of the Ks or even have access to a trusted chip* to perform the write, System must follow a non-authenticated write by an authenticated read if it needs to know that the write was successful.

The protocol requires the following publicly available function:

Write[t, X] Writes X over those parts of M_t subject to P_t and the existing value for M.

To authenticate a write of M_{new} to ChipA's memory M:

a. System calls ChipU's Write function, passing in M_{new};

b. The authentication procedure for a Read is carried out (see Section 9.3 on page 41);

c. If ChipU is authentic and M_{new} = M returned in **b**, the write succeeded. If not, it failed.

10.4.2 Authenticated writes

In the multiple memory vectors protocol, only M_0 can be written to an an authenticated way. This is because only M_0 is considered to have components that need to be upgraded.

In this kind of write, System wants to change Chip U's M_0 in an authorized way, without being subject to the permissions that apply during normal operation. For example, the consumable may be at a refilling station and the normally Decrement Only section of M_0 should be updated to include the new valid consumable. In this case, the chip whose M_0 is being updated must authenticate the writes being generated by the external System and in addition, apply the appropriate permission for the key to ensure that only the correct parts of M_0 are updated. Having a different permission for each key is required as when multiple keys are involved, all keys should not necessarily be given open access to M_0. For example, suppose M_0 contains printer speed and a counter of money available for franking. A ChipS that updates printer speed should not be capable of updating the amount of money. Since $P_{0...T-1}$ is used for non-authenticated writes, each K_n has a corresponding permission P_{T+n} that determines what can be updated in an authenticated write.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Read[n, t, X] Advances R, and returns R, M_t, $S_{Kn}[X|R|C_1|M_t]$. The time taken to calculate the signature must not be based on the contents of X, R, M_t, or K.

WriteA[n, X, Y, Z] Advances R, replaces M_0 by Y subject to P_{T+n}, and returns 1 only if $S_{Kn}[R|X|C_1|Y] = Z$. Otherwise returns 0. The time taken to calculate and compare signatures must be independent of data content. This function is identical to ChipT's Test function except that it additionally writes Y subject to P_{T+n} to its M when the signature matches.

Authenticated writes require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of M that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be

updated by another ChipS that will perform updates to that part of M (assuming ChipS's P allows that part of M to be updated).

Q Part of M that contains the write permissions for updating ChipU's M. By adding Q to ChipS we allow different ChipSs that can update different parts of M_U. Permissions in ChipS's $P_{0..T-1}$ for this part of M needs to be ReadOnly once ChipS has been setup. Therefore Q can only be updated by another ChipS that will perform updates to that part of M.

SignM[n,V,W,X,Y,Z] Advances R, decrements CountRemaining and returns R, Z_{QX} (Z applied to X with permissions Q), $S_{Kn}[W|R|C_1|Z_{QX}]$ only if Y = $S_{Kn}[V|W|C_1|X]$ and CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's M vector:

a. System calls ChipU's Read function, passing in n1, 0 and 0 as the input parameters;

b. ChipU produces R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$ and returns these to System;

c. System calls ChipS's SignM function, passing in n2 (the key to be used in ChipS), 0 (as used in **a**), R_U, M_{U0}, $S_{Kn1}[0|R_U|C_1|M_{U0}]$, and M_D (the desired vector to be written to ChipU);

d. ChipS produces R_S, M_{QD} (processed by running M_D against M_{U0} using Q) and $S_{Kn2}[R_U|R_S|C_1|M_{QD}]$ if the inputs were valid, and 0 for all outputs if the inputs were not valid.

e. If values returned in **d** are non zero, then ChipU is considered authentic. System can then call ChipU's WriteA function with these values from **d**.

f. ChipU should return a 1 to indicate success. A 0 should only be returned if the data generated by ChipS is incorrect (e.g. a transmission error).

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes is shown in Figure 6:

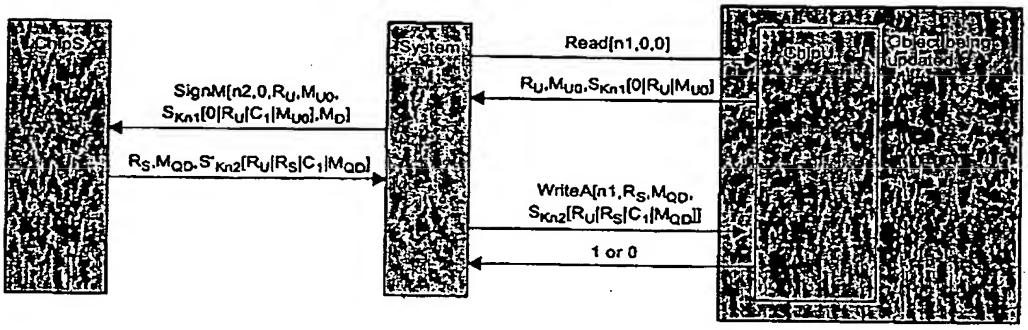


Figure 10. Protocol for multiple key authenticated write

Note that Q in ChipS is part of ChipS's M. This allows a user to set up ChipS with a permission set for upgrades. This should be done to ChipS and that part of M designated by $P_{0..T-1}$ set to ReadOnly *before* ChipS is programmed with K_U. If K_S is programmed with K_U first, there is a risk of someone obtaining a half-setup ChipS and changing all of M_U instead of only the sections specified by Q.

In addition, CountRemaining in ChipS needs to be setup (including making it ReadOnly in P_S) before ChipS is programmed with K_U. ChipS should therefore be programmed to only perform a limited number of SignM operations (thereby limiting compromise exposure if a ChipS is stolen). Thus ChipS would itself need to be upgraded with a new CountRemaining every so often.

10.4.3 Updating permissions for future writes

In order to reduce exposure to accidental and malicious attacks on P (and certain parts of M), only authorized users are allowed to update P. Writes to P are the same as authorized writes to M, except that they update P_n instead of M. Initially (at manufacture), P is set to be Read/Write for all M. As different processes fill up different parts of M, they can be sealed against future change by updating the permissions. Updating a chip's $P_{0..T-1}$ changes permissions for unauthorized writes to M_n, and updating $P_{T..T+N-1}$ changes permissions for authorized writes with key K_n.

P_n is only allowed to change to be a more restrictive form of itself. For example, initially all parts of M have permissions of Read/Write. A permission of Read/Write can be updated to Decrement Only or Read Only. A permission of Decrement Only can be updated to become Read Only. A Read Only permission cannot be further restricted.

In this transaction protocol, the System's chip is referred to as ChipS, and the chip being updated is referred to as ChipU. Each chip distrusts the other.

The protocol requires the following publicly available functions in ChipU:

Random[] Returns R (does not advance R).

SetPermission[n,p,X,Y,Z] Advances R, and updates P_p according to Y and returns 1 followed by the resultant P_p only if $S_{Kn}[R|X|Y|C_2] = Z$. Otherwise returns 0. P_p can only become more restricted. Passing in 0 for any permission leaves it unchanged (passing in Y=0 returns the current P_p).

Authenticated writes of permissions require that the System has access to a ChipS that is capable of generating appropriate signatures. ChipS requires the following variables and function:

CountRemaining Part of ChipS's M_0 that contains the number of signatures that ChipS is allowed to generate. Decrements with each successful call to SignM and SignP. Permissions in ChipS's $P_{0..T-1}$ for this part of M_0 needs to be ReadOnly once ChipS has been setup. Therefore CountRemaining can only be updated by another ChipS that will perform updates to that part of M_0 (assuming ChipS's P_n allows that part of M_0 to be updated).

SignP[n,X,Y] Advances R, decrements CountRemaining and returns R and $S_{Kn}[X|R|Y|C_2]$ only if CountRemaining > 0. Otherwise returns all 0s. The time taken to calculate and compare signatures must be independent of data content.

To update ChipU's P_n:

a. System calls ChipU's Random function;

b. ChipU returns R_U to System;

c. System calls ChipS's SignP function, passing in n1, R_U and P_D (the desired P to be written to ChipU);

d. ChipS produces R_S and $S_{Kn1}[R_U|R_S|P_D|C_2]$ if it is still permitted to produce signatures.

e. If values returned in **d** are non zero, then System can then call ChipU's SetPermission function with n2, the desired permission entry p, R_S, P_D and $S_{Kn1}[R_U|R_S|P_D|C_2]$.

f. ChipU verifies the received signature against $S_{Kn2}[R_U|R_S|P_D|C_2]$ and applies P_D to P_n if the signature matches

g. System checks 1st output parameter. 1 = success, 0 = failure.

The choice of n1 and n2 must be such that ChipU's K_{n1} = ChipS's K_{n2}.

The data flow for authenticated writes to permissions is shown in Figure 7:

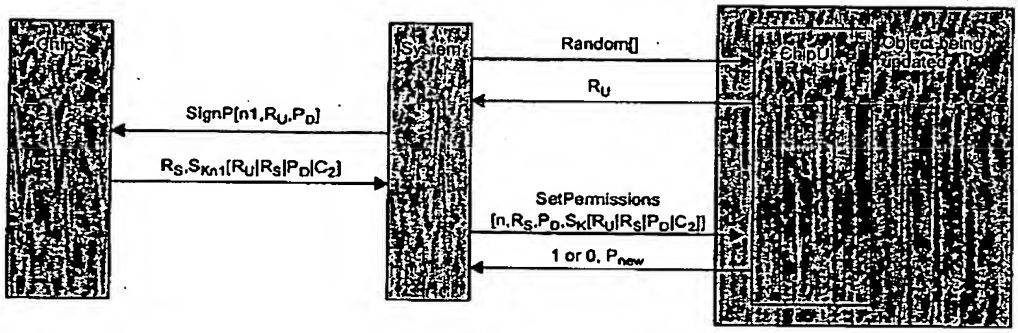


Figure 11. Protocol for multiple key update of permissions

10.4.4 Protecting M in a multiple key multiple M system

To protect the appropriate part of M_n against unauthorized writes, call SetPermissions[n] for n = 0 to T-1. To protect the appropriate part of M_0 against authorized writes with key n, call SetPermissions[T+n] for n=0 to N-1.

Note that only M_0 can be written in an authenticated fashion.

Note that the SetPermission function must be called *after* the part of M has been set to the desired value.

For example, if adding a serial number to an area of M_1 that is currently ReadWrite so that noone is permitted to update the number again:

- the Write function is called to write the serial number to M_1
- SetPermission(1) is called for to set that part of M to be ReadOnly for non-authorized writes.

If adding a consumable value to M_0 such that only keys 1-2 can update it, and keys 0, and 3-N cannot:

- the Write function is called to write the amount of consumable to M
- SetPermission is called for 0 to set that part of M_0 to be DecrementOnly for *non-authorized* writes. This allows the amount of consumable to decrement.
- SetPermission is called for n = {T, T+3, T+4 ..., T+N-1} to set that part of M_0 to be ReadOnly for *authorized* writes using all but keys 1 and 2. This leaves keys 1 and 2 with ReadWrite permissions to M_0.

It is possible for someone who knows a key to further restrict other keys, but it is not in anyone's interest to do so.

10.5 PROGRAMMING K

This section is identical to the multiple key single memory vector (Section 9.5 on page 46). It is repeated here with mention to M_0 instead of M for CountRemaining.

In this case, we have a factory chip (*ChipF*) connected to a System. The System wants to program the key in another chip (*ChipP*). System wants to avoid passing the new key to ChipP in the clear, and also wants to avoid the possibility of the key-upgrade message being replayed on another ChipP (even if the user doesn't know the key).

The protocol is a simple extension of the single key protocol in that it assumes that ChipF and ChipP already share a secret key K_{old}. This key is used to ensure that only a chip that knows K_{old} can set K_{new}.

The protocol requires the following publicly available functions in ChipP:

Random[] Returns R (does not advance R).

ReplaceKey[n, X, Y, Z] Replaces K_n by $S_{Kn}[R|X|C_3]\oplus Y$, advances R, and returns 1 only if $S_{Kn}[X|Y|C_3] = Z$. Otherwise returns 0. The time taken to calculate signatures and compare values must be identical for all inputs.

And the following data and functions in ChipF:

CountRemaining Part of M_0 with contains the number of signatures that ChipF is allowed to generate. Decrements with each successful call to GetProgramKey. Permissions in P for this part of M_0 needs to be ReadOnly once ChipF has been setup. Therefore can only be updated by a ChipS that has authority to perform updates to that part of M_0.

K_{new} The new key to be transferred from ChipF to ChipP. Must not be visible.

SetPartialKey[X,Y] If word X of K_{new} has not yet been set, set word X of K_{new} to Y and return 1. Otherwise return 0. This function allows K_{new} to be programmed in multiple steps, thereby allowing different people or systems to know different parts of the key (but not the whole K_{new}). K_{new} is stored in ChipF's flash memory. Since there is a small number of ChipFs, it is theoretically not necessary to store the inverse of K_{new}, but it is stronger protection to do so.

GetProgramKey[n, X] Advances R_F, decrements CountRemaining, outputs R_F, the encrypted key $S_{Kn}[X|R_F|C_3]\oplus K_{new}$ and a signature of the first two outputs plus C_3 if CountRemaining>0. Otherwise outputs 0. The time to calculate the encrypted key & signature must be identical for all inputs.

To update P's key :

a. System calls ChipP's Random function;

b. ChipP returns R_P to System;

c. System calls ChipF's GetProgramKey function, passing in n1 (the desired key to use) and the result from **b**;

d. ChipF updates R_F, then calculates and returns R_F, $S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}$, and $S_{Kn1}[R_F|S_{Kn1}[R_P|R_F|C_3]\oplus K_{new}|C_3]$;

e. If the response from **d** is not 0, System calls ChipP's ReplaceKey function, passing in n2 (the key to use in ChipP) and the response from **d**;

f. System checks response from ChipP. If the response is 1, then K_{Pn2} has been correctly updated to K_{new}. If the response is 0, K_{Pn2} has not been updated.

The choice of n1 and n2 must be such that ChipF's K_{n1} = ChipP's K_{n2}.

The data flow for key updates is shown in Figure 8:

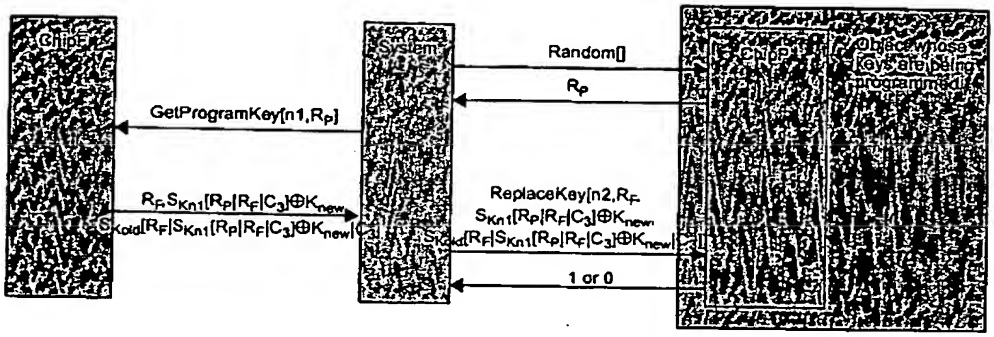


Figure 12. Protocol for multiple key update

Note that K_{new} is never passed in the open. An attacker could send its own R_P, but cannot produce $S_{Kn1}[R_P|R_F|C_3]$ without K_{n1}. The signature based on K_{new} is sent to ensure that ChipP will be able to determine if either of the first two parameters have been changed en route.

CountRemaining needs to be setup in M_{F0} (including making it ReadOnly in P) before ChipF is programmed with K_P. ChipF should therefore be programmed to only perform a limited number of GetProgramKey operations (thereby limiting compromise exposure if a ChipF is stolen). An authorized ChipS can be used to update this counter if neccesary (see Section 9.4 on page 42).

10.5.1 Chicken and Egg

As with the single key protocol, for the Program Key protocol to work, both ChipF and ChipP must both know K_{old}. Obviously both chips had to be programmed with K_{old}, and thus K_{old} can be thought of as an older K_{new}: K_{old} can be placed in chips if another ChipF knows K_{older}, and so on.

Although this process allows a chain of reprogramming of keys, with each stage secure, at some stage the very first key (K_{first}) must be placed in the chips. K_{first} is in fact programmed with the chip's microcode at the manufacturing test station as the last step in manufacturing test. K_{first} can be a manufacturing batch key, changed for each batch or for each customer etc, and can have as short a life as desired. Compromising K_{first} need not result in a complete compromise of the chain of Ks.

Depending on reprogramming requirements, K_{first} can be the same or different for all K_n.

10.5.2 Security Note

It is imperative that different ChipFs have different R_F values to prevent K_{new} from being determined as follows:

The attacker needs 2 ChipFs, both with the same R_F and K_n but different values for K_{new}. By knowing K_{new1} the attacker can determine K_{new2}. The size of R_F is 2^{160}, and assuming a lifespan of approximately 2^{32} Rs, an attacker needs about 2^{60} ChipFs with the same K_n to locate the correct chip. Given that there are likely to be only hundreds of ChipFs with the same K_n, this is not a likely attack. The attack can be eliminated completely by making C_3 different per chip and transmitting it with the new signature.

11 Summary of functions for all protocols

All protocol sets, whether single key, multiple key, single M or multiple M, all rely on the same set of functions. The function set is listed here:

11.1 ALL CHIPS

Since every chip must act as ChipP, ChipA and potentially ChipU, *all* chips require the following functions:

- Random
- ReplaceKey
- Read
- Write
- WriteA
- SetPermissions

11.2 CHIPT

Chips that are to be used as ChipT also require:

- Test

11.3 CHIPS

Chips that are to be used as ChipS also require either or both of:

- SignM
- SignP

11.4 CHIPF

Chips that are to be used as ChipF also require:

- SetPartialKey
- GetProgramKey

12 Remote Upgrades

12.1 BASIC REMOTE UPGRADES

Regardless of the number of keys and the number of memory vectors, the use of authenticated reads and writes, and of replacing a new key without revealing K_{new} or K_{old} allows the possibility of remote upgrades of ChipU and ChipP. The upgrade typically involves a remote server and follows two basic steps:

a. During the first stage of the upgrade, the remote system authenticates the user's system to ensure the user's system has the setup that it claims to have.

b. During the second stage of the upgrade, the user's system authenticates the remote system to ensure that the upgrade is from a trusted source.

12.1.1 User requests upgrade

The user requests that he wants to upgrade. This can be done by running a specific upgrade application on the user's computer, or by visiting a specific website.

12.1.2 Remote system gathers info securely about user's current setup

In this step, the remote system determines the current setup for the user. The current setup must be authenticated, to ensure that the user truly has the setup that is claimed. Traditionally, this has been by checking the existence of files, generating checksums from those files, or by getting a serial number from a hardware dongle, although these traditional methods have difficulties since they can be generated locally by "hacked" software.

The authenticated read protocol described in Section 8.3 on page 34 can be used to accomplish this step. The use of random numbers has the advantage that the local user cannot capture a successful transaction and play it back on another computer system to fool the remote system.

12.1.3 Remote system gives user choice of upgrade possibilities & user chooses

If there is more than one upgrade possibility, the various upgrade options are now presented to the user. The upgrade options could vary based on a number of factors, including, but not limited to:

- current user setup
- user's preference for payment schemes (e.g. single payment vs. multiple payment)
- number of other products owned by user

The user selects an appropriate upgrade and pays if necessary (by some scheme such as via a secure web site). What is important to note here is that the user chooses a specific upgrade and commences the upgrade operation.

12.1.4 Remote system sends upgrade request to local system

The remote system now instructs the local system to perform the upgrade. However, the local system can only accept an upgrade from the remote system if the remote system is also authenticated. This is effectively an authenticated write. The use of R_U in the signature prevents the upgrade message from being replayed on another ChipU.

If multiple keys are used, and each chip has a unique key, the remote system can use a serial number obtained from the current setup (authenticated by a common key) to lookup

the unique key for use in the upgrade. Although the random number provides time varying messages, use of an unknown K that is different for each chip means that collection and examination of messages and their signatures is made even more difficult.

12.2 OEM UPGRADES

OEM upgrades are effectively the same as remote upgrades, except that the user interacts with an OEM server for upgrade selection. The OEM server may send sub-requests to the manufacturer's remote server to provide authentication, upgrade availability lists, and base-level pricing information.

An additional level of authentication may be incorporated into the protocol to ensure that upgrade requests are coming from the OEM server, and not from a 3rd party. This can readily be incorporated into both authentication steps.

13 Choice of Signature Function

Given that all protocols make use of keyed signature functions, the choice of function is examined here.

Table 2 outlines the attributes of the applicable choices (see Section 5.2 on page 5 and Section 5.5 on page 11 for more information). The attributes are phrased so that the attribute is seen as an advantage.

Table 2. Attributes of Applicable Signature Functions

	Triple DES	Blowfish	RC5	IDEA	Random Sequence	HMAC-MD5	HMAC-SHA1	HMAC-RIPEMD160
Free of patents	•	•			•	•	•	•
Random key generation						•	•	•
Can be exported from the USA					•	•	•	•
Fast		•				•	•	•
Preferred Key Size (bits) for use in this application	168[a]	128	128	128	512	128	160	160
Block size (bits)	64	64	64	64	256	512	512	512
Cryptanalysis Attack-Free (apart from weak keys)	•	•			•		•	•
Output size given input size N	≥N	≥N	≥N	≥N	128	128	160	160
Low storage requirements					•	•	•	•
Low silicon complexity					•	•	•	•
NSA designed	•						•	

a. Only gives protection equivalent to 112-bit DES

An examination of Table 2 shows that the choice is effectively between the 3 HMAC constructs and the Random Sequence. The problem of key size and key generation eliminates the Random Sequence. Given that a number of attacks have already been carried out on MD5 and since the hash result is only 128 bits, HMAC-MD5 is also eliminated. The choice is therefore between HMAC-SHA1 and HMAC-RIPEMD160.

RIPEMD-160 is relatively new, and has not been as extensively cryptanalyzed as SHA-1. However, SHA-1 was designed by the NSA, so this may be seen by some as a negative attribute. According to Schneier [80] in a comparison of relative security of hash algorithms:

> *"I recommend SHA-1, from the National Security Agency (NSA)... Another choice is RIPE-MD-160.*
>
> *Not much is forthcoming in this category. A lot of work has been done on creating hash functions from block ciphers, but no single proposal has emerged as a front-runner. People are likely to stick with SHA-1 or RIPE-MD-160..."*

Given that there is not much between the two, I recommend SHA-1 for the HMAC construct for the following reasons:

- SHA-1 was designed by the NSA;
- SHA-1 has been more extensively cryptanalyzed without being broken;
- SHA-1 requires slightly less intermediate storage than RIPE-MD-160;
- SHA-1 is algorithmically less complex than RIPE-MD-160;

Although SHA-1 is slightly faster than RIPE-MD-160, this was not a reason for choosing SHA-1.

13.1 HMAC-SHA1

The mechanism for authentication is the HMAC-SHA1 algorithm. This section examines the HMAC-SHA1 algorithm in greater detail than covered so far, and describes an optimization of the algorithm that requires fewer memory resources than the original definition.

13.1.1 HMAC

Given the following definitions:

- H = the hash function (e.g. MD5 or SHA-1)
- n = number of bits output from H (e.g. 160 for SHA-1, 128 bits for MD5)
- M = the data to which the MAC function is to be applied
- K = the secret key shared by the two parties
- ipad = 0x36 repeated 64 times
- opad = 0x5C repeated 64 times

The HMAC algorithm is as follows:

a. Extend K to 64 bytes by appending 0x00 bytes to the end of K
b. XOR the 64 byte string created in (1) with ipad
c. append data stream M to the 64 byte string created in (2)
d. Apply H to the stream generated in (3)
e. XOR the 64 byte string created in (1) with opad
f. Append the H result from (4) to the 64 byte string resulting from (5)
g. Apply H to the output of (6) and output the result

Thus:

$$HMAC[M] = H[(K \oplus opad) \mid H[(K \oplus ipad) \mid M]]$$

The HMAC-SHA1 algorithm is simply HMAC with H = SHA-1.

13.1.2 SHA-1

The SHA1 hashing algorithm is described in the context of other hashing algorithms in Section 5.5.3.3 on page 14, and completely defined in [28]. The algorithm is summarized here.

Nine 32-bit constants are defined in Table 3. There are 5 constants used to initialize the chaining variables, and there are 4 additive constants.

Table 3. Constants used in SHA-1

Initial Chaining Values		Additive Constants	
h_1	0x67452301	y_1	0x5A827999
h_2	0xEFCDAB89	y_2	0x6ED9EBA1
h_3	0x98BADCFE	y_3	0x8F1BBCDC
h_4	0x10325476	y_4	0xCA62C1D6
h_5	0xC3D2E1F0		

Non-optimized SHA-1 requires a total of 2912 bits of data storage:

- Five 32-bit chaining variables are defined: H_1, H_2, H_3, H_4 and H_5.
- Five 32-bit working variables are defined: A, B, C, D, and E.
- One 32-bit temporary variable is defined: t.
- Eighty 32-bit temporary registers are defined: $X_{0\text{-}79}$.

The following functions are defined for SHA-1:

Table 4. Functions used in SHA-1

Symbolic Nomenclature	Description
+	Addition modulo 2^{32}
X « Y	Result of rotating X left through Y bit positions
f(X, Y, Z)	$(X \wedge Y) \vee (\neg X \wedge Z)$
g(X, Y, Z)	$(X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$
h(X, Y, Z)	$X \oplus Y \oplus Z$

The hashing algorithm consists of firstly padding the input message to be a multiple of 512 bits and initializing the chaining variables $H_{1\text{-}5}$ with $h_{1\text{-}5}$. The padded message is then processed in 512-bit chunks, with the output hash value being the final 160-bit value given by the concatenation of the chaining variables: $H_1 \mid H_2 \mid H_3 \mid H_4 \mid H_5$.

The steps of the SHA-1 algorithm are now examined in greater detail.

13.1.2.1 Step 1. Preprocessing

The first step of SHA-1 is to pad the input message to be a multiple of 512 bits as follows and to initialize the chaining variables.

Table 5. Steps to follow to preprocess the input message

Pad the input message	Append a 1 bit to the message
	Append 0 bits such that the length of the padded message is 64-bits short of a multiple of 512 bits.
	Append a 64-bit value containing the length in bits of the original input message. Store the length as most significant bit through to least significant bit.
Initialize the chaining variables	$H_1 \leftarrow h_1$, $H_2 \leftarrow h_2$, $H_3 \leftarrow h_3$, $H_4 \leftarrow h_4$, $H_5 \leftarrow h_5$

13.1.2.2 Step 2. Processing

The padded input message can now be processed.

We process the message in 512-bit blocks. Each 512-bit block is in the form of 16 × 32-bit words, referred to as $InputWord_{0-15}$.

Table 6. Steps to follow for each 512 bit block ($InputWord_{0-15}$)

Copy the 512 input bits into X_{0-15}	For j=0 to 15 $X_j = InputWord_j$
Expand X_{0-15} into X_{16-79}	For j=16 to 79 $Xj \leftarrow ((X_{j-3} \oplus X_{j-8} \oplus X_{j-14} \oplus X_{j-16}) \ll 1)$
Initialize working variables	$A \leftarrow H_1$, $B \leftarrow H_2$, $C \leftarrow H_3$, $D \leftarrow H_4$, $E \leftarrow H_5$
Round 1	For j=0 to 19 $t \leftarrow ((A \ll 5) + f(B, C, D) + E + Xj + y_1)$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \ll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 2	For j=20 to 39 $t \leftarrow ((A \ll 5) + h(B, C, D) + E + Xj + y_2)$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \ll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 3	For j=40 to 59 $t \leftarrow ((A \ll 5) + g(B, C, D) + E + Xj + y_3)$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \ll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 4	For j=60 to 79 $t \leftarrow ((A \ll 5) + h(B, C, D) + E + Xj + y_4)$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \ll 30)$, $B \leftarrow A$, $A \leftarrow t$
Update chaining variables	$H_1 \leftarrow H_1 + A$, $H_2 \leftarrow H_2 + B$, $H_3 \leftarrow H_3 + C$, $H_4 \leftarrow H_4 + D$, $H_5 \leftarrow H_5 + E$

The bold text is to emphasize the differences between each round.

13.1.2.3 Step 3. Completion

After all the 512-bit blocks of the padded input message have been processed, the output hash value is the final 160-bit value given by: $H_1 \mid H_2 \mid H_3 \mid H_4 \mid H_5$.

13.1.2.4 Optimization for hardware implementation

The SHA-1 Step 2 procedure is not optimized for hardware. In particular, the 80 temporary 32-bit registers use up valuable silicon on a hardware implementation. This section describes an optimization to the SHA-1 algorithm that only uses 16 temporary registers. The reduction in silicon is from 2560 bits down to 512 bits, a saving of over 2000 bits. It may not be important in some applications, but in the QA Chip storage space must be reduced where possible.

The optimization is based on the fact that although the original 16-word message block is expanded into an 80-word message block, the 80 words are not updated during the algorithm. In addition, the words rely on the previous 16 words only, and hence the expanded words can be calculated on-the-fly during processing, as long as we keep 16 words for the backward references. We require rotating counters to keep track of which register we are up to using, but the effect is to save a large amount of storage.

Rather than index X by a single value j, we use a 5 bit counter to count through the iterations. This can be achieved by initializing a 5-bit register with either 16 or 20, and decrementing it until it reaches 0. In order to update the 16 temporary variables as if they were

80, we require 4 indexes, each a 4-bit register. All 4 indexes increment (with wraparound) during the course of the algorithm.

Table 7. Optimised Steps to follow for each 512 bit block ($InputWord_{0-15}$)

Initialize working variables	$A \leftarrow H_1$, $B \leftarrow H_2$, $C \leftarrow H_3$, $D \leftarrow H_4$, $E \leftarrow H_5$ $N_1 \leftarrow 13$, $N_2 \leftarrow 8$, $N_3 \leftarrow 2$, $N_4 \leftarrow 0$
Round 0 Copy the 512 input bits into X_{0-15}	Do 16 times $X_{N4} = InputWord_{N4}$ $[\Uparrow N_1, \Uparrow N_2, \Uparrow N_3]_{optional}$ $\Uparrow N_4$
Round 1A	Do 16 times $t \leftarrow ((A \lll 5) + f(B, C, D) + E + X_{N4} + y_1)$ $[\Uparrow N_1, \Uparrow N_2, \Uparrow N_3]_{optional}$ $\Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \lll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 1B	Do 4 times $X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \lll 1)$ $t \leftarrow ((A \lll 5) + f(B, C, D) + E + X_{N4} + y_1)$ $\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \lll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 2	Do 20 times $X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \lll 1)$ $t \leftarrow ((A \lll 5) + h(B, C, D) + E + X_{N4} + y_2)$ $\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \lll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 3	Do 20 times $X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \lll 1)$ $t \leftarrow ((A \lll 5) + g(B, C, D) + E + X_{N4} + y_3)$ $\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \lll 30)$, $B \leftarrow A$, $A \leftarrow t$
Round 4	Do 20 times $X_{N4} \leftarrow ((X_{N1} \oplus X_{N2} \oplus X_{N3} \oplus X_{N4}) \lll 1)$ $t \leftarrow ((A \lll 5) + h(B, C, D) + E + X_{N4} + y_4)$ $\Uparrow N_1, \Uparrow N_2, \Uparrow N_3, \Uparrow N_4$ $E \leftarrow D$, $D \leftarrow C$, $C \leftarrow (B \lll 30)$, $B \leftarrow A$, $A \leftarrow t$
Update chaining variables	$H_1 \leftarrow H_1 + A$, $H_2 \leftarrow H_2 + B$, $H_3 \leftarrow H_3 + C$, $H_4 \leftarrow H_4 + D$, $H_5 \leftarrow H_5 + E$

The bold text is to emphasize the differences between each round.

The incrementing of N_1, N_2, and N_3 during Rounds 0 and 1A is optional. A software implementation would not increment them, since it takes time, and at the end of the 16 times through the loop, all 4 counters will be their original values. Designers of hardware may wish to increment all 4 counters together to save on control logic.

Round 0 can be completely omitted if the caller loads the 512 bits of X_{0-15}.

14 Holding Out Against Attacks

The authentication protocols described in Section 7 on page 32 onward should be resistant to defeat by logical means. This section details each type of attack in turn with reference to the Read Authentication protocol.

14.1 BRUTE FORCE ATTACK

A brute force attack is guaranteed to break any protocol. However the length of the key means that the time for an attacker to perform a brute force attack is too long to be worth the effort.

An attacker only needs to break K to build a clone authentication chip. A brute force attack on K must therefore break a 160-bit key.

An attack against K requires a maximum of 2^{160} attempts, with a 50% chance of finding the key after only 2^{159} attempts. Assuming an array of a trillion processors, each running one million tests per second, 2^{159} (7.3×10^{47}) tests takes 2.3×10^{22} years, which is longer than the total lifetime of the universe. There are around 100 million personal computers in the world. Even if these were all connected in an attack (e.g. via the Internet), this number is still 10,000 times smaller than the trillion-processor attack described. Further, if the manufacture of one trillion processors becomes a possibility in the age of nanocomputers, the time taken to obtain the key is still longer than the total lifetime of the universe.

14.2 GUESSING THE KEY ATTACK

It is theoretically possible that an attacker can simply "guess the key". In fact, given enough time, and trying every possible number, an attacker will obtain the key. This is identical to the brute force attack described above, where 2^{159} attempts must be made before a 50% chance of success is obtained.

The chances of someone simply guessing the key on the first try is 2^{160}. For comparison, the chance of someone winning the top prize in a U.S. state lottery and being killed by lightning in the same day is only 1 in 2^{61} [78]. The chance of someone guessing the authentication chip key on the first go is 1 in 2^{160}, which is comparable to two people choosing exactly the same atoms from a choice of all the atoms in the Earth i.e. extremely unlikely.

14.3 QUANTUM COMPUTER ATTACK

To break K, a quantum computer containing 160 qubits embedded in an appropriate algorithm must be built. As described in Section 5.7.1.7 on page 20, an attack against a 160-bit key is not feasible. An outside estimate of the possibility of quantum computers is that 50 qubits may be achievable within 50 years. Even using a 50 qubit quantum computer, 2^{110} tests are required to crack a 160 bit key. Assuming an array of 1 billion 50 qubit quantum computers, each able to try 2^{50} keys in 1 microsecond (beyond the current wildest estimates) finding the key would take an average of 18 billion years.

14.4 CIPHERTEXT ONLY ATTACK

An attacker can launch a ciphertext only attack on K by monitoring calls to Random and Read. However, given that all these calls also reveal the plaintext as well as the hashed

form of the plaintext, the attack would be transformed into a stronger form of attack - a known plaintext attack.

14.5 KNOWN PLAINTEXT ATTACK

It is easy to connect a logic analyzer to the connection between the System and the authentication chip, and thereby monitor the flow of data. This flow of data results in known plaintext and the hashed form of the plaintext, which can therefore be used to launch a known plaintext attack against K.

To launch an attack against K, multiple calls to Random and Test must be made (with the call to Test being successful, and therefore requiring a call to Read on a valid chip). This is straightforward, requiring the attacker to have both a system authentication chip and a consumable authentication chip. For each set of calls, an X, $S_K[X]$ pair is revealed. The attacker must collect these pairs for further analysis.

The question arises of how many pairs must be collected for a meaningful attack to be launched with this data. An example of an attack that requires collection of data for statistical analysis is differential cryptanalysis (see Section 14.13 on page 70). However, there are no known attacks against SHA-1 or HMAC-SHA1 [7][56][78], so there is no use for the collected data at this time.

14.6 CHOSEN PLAINTEXT ATTACKS

The golden rule for the QA Chip is that it never signs something that is simply given to it - i.e. it never lets the user choose the message that is signed.

Although the attacker can choose both R_T and possibly M, ChipA advances its random number R_A with each call to Read. The resultant message X therefore contains 160 bits of changing data each call that are not chosen by the attacker.

To launch a chosen text attack the attacker would need to locate a chip whose R was the desired R. This makes the search effectively impossible.

14.7 ADAPTIVE CHOSEN PLAINTEXT ATTACKS

The HMAC construct provides security against all forms of chosen plaintext attacks [7]. This is primarily because the HMAC construct has 2 secret input variables (the result of the original hash, and the secret key). Thus finding collisions in the hash function itself when the input variable is secret is even harder than finding collisions in the plain hash function. This is because the former requires direct access to SHA-1 in order to generate pairs of input/output from SHA-1.

Since R changes with each call to Read, the user cannot choose the complete message. The only value that can be collected by an attacker is $HMAC[R_1 \mid R_2 \mid M_2]$. These are not attacks against the SHA-1 hash function itself, and reduce the attack to a differential cryptanalysis attack (see Section 14.13 on page 70), examining statistical differences between collected data. Given that there is no differential cryptanalysis attack known against SHA-1 or HMAC, the protocols are resistant to the adaptive chosen plaintext attacks.

14.8 PURPOSEFUL ERROR ATTACK

An attacker can only launch a purposeful error attack on the Test function, since this is the only function in the Read protocol that validates input against the keys.

With the Test function, a 0 value is produced if an error is found in the input - no further information is given. In addition, the time taken to produce the 0 result is independent of the input, giving the attacker no information about which bit(s) were wrong.

A purposeful error attack is therefore fruitless.

14.9 CHAINING ATTACK

Any form of chaining attack assumes that the message to be hashed is over several blocks, or the input variables can somehow be set. The HMAC-SHA1 algorithm used by Protocol C1 only ever hashes one or two 512-bit blocks. Chaining attacks are not possible when only one block is used, and are extremely limited when two blocks are used.

14.10 BIRTHDAY ATTACK

The strongest attack known against HMAC is the birthday attack, based on the frequency of collisions for the hash function [7][51]. However this is totally impractical for minimally reasonable hash functions such as SHA-1. And the birthday attack is only possible when the attacker has control over the message that is hashed.

Since in the protocols described for the QA Chip, the message to be signed is never chosen by the attacker (at least one 160-bit R value is chosen by the chip doing the signing), the attacker has no control over the message that is hashed. An attacker must instead search for a collision message that hashes to the same value (analogous to finding one person who shares your birthday).

The clone chip must therefore attempt to find a new value R_2 such that the hash of R_1, R_2 and a chosen M_2 yields the same hash value as $H[R_1|R_2|M]$. However ChipT does not reveal the correct hash value (the Test function only returns 1 or 0 depending on whether the hash value is correct). Therefore the only way of finding out the correct hash value (in order to find a collision) is to interrogate a real ChipA. But to find the correct value means to update M, and since the decrement-only parts of M are one-way, and the read-only parts of M cannot be changed, a clone consumable would have to update a real consumable before attempting to find a collision. The alternative is a brute force attack search on the Test function to find a success (requiring each clone consumable to have access to a System consumable). A brute force search, as described above, takes longer than the lifetime of the universe, in this case, per authentication.

There is no point for a clone consumable to launch this kind of attack.

14.11 SUBSTITUTION WITH A COMPLETE LOOKUP TABLE

The random number seed in each System is 160 bits. The best case situation for an attacker is that no state data has been changed. Assuming also that the clone consumable does not advance its R, there is a constant value returned as M. A clone chip must therefore return $S_K[R \mid c]$ (where c is a constant), which is a 160 bit value.

Assuming a 160-bit lookup of a 160-bit result, this requires 2.9×10^{49} bytes, or 2.6×10^{37} terabytes, certainly more space than is feasible for the near future. This of course does not even take into account the method of collecting the values for the ROM. A complete lookup table is therefore completely impossible.

14.12 SUBSTITUTION WITH A SPARSE LOOKUP TABLE

A sparse lookup table is only feasible if the messages sent to the authentication chip are somehow predictable, rather than effectively random.

The random number R is seeded with an unknown random number, gathered from a naturally random event. There is no possibility for a clone manufacturer to know what the possible range of R is for all Systems, since each bit has an unrelated chance of being 1 or 0.

Since the range of R in all systems is unknown, it is not possible to build a sparse lookup table that can be used in all systems. The general sparse lookup table is therefore not a possible attack.

However, it is possible for a clone manufacturer to know what the range of R is for a given System. This can be accomplished by loading a LFSR with the current result from a call to a specific System authentication chip's Random function, and iterating some number of times into the future. If this is done, a special ROM can be built which will only contain the responses for that particular range of R, i.e. a ROM specifically for the consumables of that particular System. But the attacker still needs to place correct information in the ROM. The attacker will therefore need to find a valid authentication chip and call it for each of the values in R.

Suppose the clone authentication chip reports a full consumable, and then allows a single use before simulating loss of connection and insertion of a new full consumable. The clone consumable would therefore need to contain responses for authentication of a full consumable and authentication of a partially used consumable. The worst case ROM contains entries for full and partially used consumables for R over the lifetime of System. However, a valid authentication chip must be used to generate the information, and be partially used in the process. If a given System only produces n R-values, the sparse lookup-ROM required is $20n$ bytes ($20 = 160 / 8$) multiplied by the number of different values for M. The time taken to build the ROM depends on the amount of time enforced between calls to Read.

After all this, the clone manufacturer must rely on the consumer returning for a refill, since the cost of building the ROM in the first place consumes a single consumable. The clone manufacturer's business in such a situation is consequently in the refills.

The time and cost then, depends on the size of R and the number of different values for M that must be incorporated in the lookup. In addition, a custom clone consumable ROM must be built to match each and every System, and a different valid authentication chip must be used for each System (in order to provide the full and partially used data). The use of an authentication chip in a System must therefore be examined to determine whether or not this kind of attack is worthwhile for a clone manufacturer.

As an example, of a camera system that has about 10,000 prints in its lifetime. Assume it has a single Decrement Only value (number of prints remaining), and a delay of 1 second between calls to Read. In such a system, the sparse table will take about 3 hours to build, and consumes 100K. Remember that the construction of the ROM requires the consump-

tion of a valid authentication chip, so any money charged must be worth more than a single consumable and the clone consumable combined. Thus it is not cost effective to perform this function for a single consumable (unless the clone consumable somehow contained the equivalent of multiple authentic consumables).

If a clone manufacturer is going to go to the trouble of building a custom ROM for each owner of a System, an easier approach would be to update System to completely ignore the authentication chip.

Consequently, this attack is possible as a per-System attack, and a decision must be made about the chance of this occurring for a given System/Consumable combination. The chance will depend on the cost of the consumable and authentication chips, the longevity of the consumable, the profit margin on the consumable, the time taken to generate the ROM, the size of the resultant ROM, and whether customers will come back to the clone manufacturer for refills that use the same clone chip etc.

14.13 DIFFERENTIAL CRYPTANALYSIS

Existing differential attacks are heavily dependent on the structure of S boxes, as used in DES and other similar algorithms. Although HMAC-SHA1 has no S boxes, an attacker can undertake a differential-like attack by undertaking statistical analysis of:

- Minimal-difference inputs, and their corresponding outputs
- Minimal-difference outputs, and their corresponding inputs

To launch an attack of this nature, sets of input/output pairs must be collected. The collection can be via known plaintext, or from a partially adaptive chosen plaintext attack. Obviously the latter, being chosen, will be more useful.

Hashing algorithms in general are designed to be resistant to differential analysis. SHA-1 in particular has been specifically strengthened, especially by the 80 word expansion so that minimal differences in input will still produce outputs that vary in a larger number of bit positions (compared to 128 bit hash functions). In addition, the information collected is not a direct SHA-1 input/output set, due to the nature of the HMAC algorithm. The HMAC algorithm hashes a known value with an unknown value (the key), and the result of this hash is then rehashed with a separate unknown value. Since the attacker does not know the secret value, nor the result of the first hash, the inputs and outputs from SHA-1 are not known, making any differential attack extremely difficult.

There are no known differential attacks against SHA-1 or HMAC-SHA-1[56][78].

The following is a more detailed discussion of minimally different inputs and outputs from the QA Chip.

14.13.1 Minimal difference inputs

This is where an attacker takes a set of X, $S_K[X]$ values where the X values are minimally different, and examines the statistical differences between the outputs $S_K[X]$. The attack relies on X values that only differ by a minimal number of bits. The question then arises as to how to obtain minimally different X values in order to compare the $S_K[X]$ values.

Although the attacker can choose both R_T and possibly M, ChipA advances its random number R_A with each call to Read. The resultant X therefore contains 160 bits of changing data each call, and is therefore not minimally different.

14.13.2 Minimal difference outputs

This is where an attacker takes a set of X, $S_K[X]$ values where the $S_K[X]$ values are minimally different, and examines the statistical differences between the X values. The attack relies on $S_K[X]$ values that only differ by a minimal number of bits.

There is no way for an attacker to generate an X value for a given $S_K[X]$. To do so would violate the fact that S is a one-way function (HMAC-SHA1). Consequently the only way for an attacker to mount an attack of this nature is to record all observed X, $S_K[X]$ pairs in a table. A search must then be made through the observed values for enough minimally different $S_K[X]$ values to undertake a statistical analysis of the X values.

14.14 MESSAGE SUBSTITUTION ATTACKS

In order for this kind of attack to be carried out, a clone consumable must contain a real authentication chip, but one that is effectively reusable since it never gets decremented. The clone authentication chip would intercept messages, and substitute its own. However this attack does not give success to the attacker.

A clone authentication chip may choose not to pass on a Write command to the real authentication chip. However the subsequent Read command must return the correct response (as if the Write had succeeded). To return the correct response, the hash value must be known for the specific R and M. An attacker can only determine the hash value by actually updating M in a real Chip, which the attacker does not want to do. Even changing the R sent by System does not help since the System authentication chip must match the R during a subsequent Test.

A message substitution attack would therefore be unsuccessful. This is only true if System updates the amount of consumable remaining before it is used.

14.15 REVERSE ENGINEERING THE KEY GENERATOR

If a pseudo-random number generator is used to generate keys, there is the potential for a clone manufacture to obtain the generator program or to deduce the random seed used. This was the way in which the security layer of the Netscape browser was initially broken [33].

14.16 BYPASSING THE AUTHENTICATION PROCESS

The System should ideally update the consumable state data before the consumable is used, and follow every write by a read (to authenticate the write). Thus each use of the consumable requires an authentication. If the System adheres to these two simple rules, a clone manufacturer will have to simulate authentication via a method above (such as sparse ROM lookup).

14.17 REUSE OF AUTHENTICATION CHIPS

Each use of the consumable requires an authentication. If a consumable has been used up, then its authentication chip will have had the appropriate state-data values decremented to 0. The chip can therefore not be used in another consumable.

Note that this only holds true for authentication chips that hold Decrement-Only data items. If there is no state data decremented with each usage, there is nothing stopping the

reuse of the chip. This is the basic difference between Presence-Only authentication and Consumable Lifetime authentication. All described protocols allow both.

The bottom line is that if a consumable has Decrement Only data items that are used by the System, the authentication chip cannot be reused without being completely reprogrammed by a valid programming station that has knowledge of the secret key (e.g. an authorized refill station).

14.18 MANAGEMENT DECISION TO OMIT AUTHENTICATION TO SAVE COSTS

Although not strictly an external attack, a decision to omit authentication in future Systems in order to save costs will have widely varying effects on different markets.

In the case of high volume consumables, it is essential to remember that it is very difficult to introduce authentication after the market has started, as systems requiring authenticated consumables will not work with older consumables still in circulation. Likewise, it is impractical to discontinue authentication at any stage, as older Systems will not work with the new, unauthenticated, consumables. In the second case, older Systems can be individually altered by replacing the System program code.

Without any form of protection, illegal cloning of high volume consumables is almost certain. However, with the patent and copyright protection, the probability of illegal cloning may be, say 50%. However, this is not the only loss possible. If a clone manufacturer were to introduce clone consumables which caused damage to the System (e.g. clogged nozzles in a printer due to poor quality ink), then the loss in market acceptance, and the expense of warranty repairs, may be significant.

In the case of a specialized pairing, such as a car/car-keys, or door/door-key, or some other similar situation, the omission of authentication in future systems is trivial and without repercussions. This is because the consumer is sold the entire set of System and Consumable authentication chips at the one time.

14.19 GARROTE/BRIBE ATTACK

If humans do not know the key, there is no amount of force or bribery that can reveal them. The use of ChipF and the ReplaceKey protocol is specifically designed to avoid the requirement of the programming station having to know the new key. However ChipF must be told the new key at some stage, and therefore it is the person(s) who enter the new key into ChipF that are at risk.

The level of security against this kind of attack is ultimately a decision for the System/Consumable owner, to be made according to the desired level of service.

For example, a car company may wish to keep a record of all keys manufactured, so that a person can request a new key to be made for their car. However this allows the potential compromise of the entire key database, allowing an attacker to make keys for any of the manufacturer's existing cars. It does not allow an attacker to make keys for any new cars. Of course, the key database itself may also be encrypted with a further key that requires a certain number of people to combine their key portions together for access. If no record is kept of which key is used in a particular car, there is no way to make additional keys should one become lost. Thus an owner will have to replace his car's authentication chip and all his car-keys. This is not necessarily a bad situation.

By contrast, in a consumable such as a printer ink cartridge, the one key combination is used for all Systems and all consumables. Certainly if no backup of the keys is kept, there is no human with knowledge of the key, and therefore no attack is possible. However, a no-backup situation is not desirable for a consumable such as ink cartridges, since if the key is lost no more consumables can be made. The manufacturer should therefore keep a backup of the key information in several parts, where a certain number of people must together combine their portions to reveal the full key information. This may be required if case the chip programming station needs to be reloaded.

In any case, none of these attacks are against the authenticated read protocol, since no humans are involved in the authentication process.

LOGICAL INTERFACE

15 Introduction

The QA Chip has a physical and a logical external interface. The physical interface defines how the QA Chip can be connected to a physical System, while the logical interface determines how that System can communicate with the QA Chip. This section deals with the logical interface.

15.1 Operating Modes

The QA Chip has four operating modes - *Idle Mode*, *Program Mode*, *Trim Mode* and *Active Mode*.

- *Idle Mode* is used to allow the chip to wait for the next instruction from the System.
- *Trim Mode* is used to determine the clock speed of the chip and to trim the frequency during the initial programming stage of the chip (when Flash memory is garbage). The clock frequency *must* be trimmed via Trim Mode *before* Program Mode is used to store the program code.
- *Program Mode* is used to load up the operating program code, and is required because the operating program code is stored in Flash memory instead of ROM (for security reasons).
- *Active Mode* is used to execute the specific authentication command specified by the System. Program code is executed in *Active Mode*. When the results of the command have been returned to the System, the chip enters *Idle Mode* to wait for the next instruction.

15.1.1 Idle Mode

The QA Chip starts up in *Idle Mode*. When the Chip is in *Idle Mode*, it waits for a command from the master by watching the primary id on the serial line.

- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Trim Mode id byte, the QA Chip enters *Trim Mode* and starts counting the number of internal clock cycles until the next byte is received.
- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Program Mode id byte, the QA Chip enters *Program Mode*.
- If the primary id matches the global id (0x00, common to all QA Chips), and the following byte from the master is the Active Mode id byte, the QA Chip enters *Active Mode* and executes startup code, allowing the chip to set itself into a state to receive authentication commands (includes setting a local id).
- If the primary id matches the chip's local id, and the following byte is a valid command code, the QA Chip enters *Active Mode*, allowing the command to be executed.

The valid 8-bit serial mode values sent after a global id are as shown in Table 8. They are specified to minimize the chances of them occurring by error after a global id (e.g. 0xFF and 0x00 are not used):

Table 8. Id byte values to place chip in specific mode

Value	Interpretation
10100101 (0xA5)	Trim Mode
10001110 (0x8E)	Program Mode
01111000 (0x78)	Active Mode

15.1.2 Trim Mode

Trim Mode is enabled by sending a global id byte (0x00) followed by the Trim Mode command byte.

The purpose of Trim Mode is to set the trim value (an internal register setting) of the internal ring oscillator so that Flash erasures and writes are of the correct duration. This is necessary due to the variation of the clock speed due to process variations. If writes an erasures are too long, the Flash memory will wear out faster than desired, and in some cases can even be damaged.

Trim Mode works by measuring the number of system clock cycles that occur inside the chip from the receipt of the Trim Mode command byte until the receipt of a data byte. When the data byte is received, the data byte is copied to the trim register and the current value of the count is transmitted to the outside world.

Once the count has been transmitted, the QA Chip returns to *Idle Mode*.

At reset, the internal trim register setting is set to a known value r. The external user can now perform the following operations:

- send the global id+write followed by the Trim Mode command byte
- send the 8-bit value v over a specified time t
- send a stop bit to signify no more data
- send the global id+read followed by the Trim Mode command byte
- receive the count c
- send a stop bit to signify no more data

At the end of this procedure, the trim register will be v, and the external user will know the relationship between external time t and internal time c. Therefore a new value for v can be calculated.

The Trim Mode procedure can be repeated a number of times, varying both t and v in known ways, measuring the resultant c. At the end of the process, the final value for v is established (and stored in the trim register for subsequent use in Program Mode). This value v must also be written to the flash for later use (every time the chip is placed in Active Mode for the first time after power-up).

15.1.3 Program Mode

Program Mode is enabled by sending a global id byte (0x00) followed by the Program Mode command byte.

The QA Chip determines whether or not the internal fuse has been blown (by reading 32-bit word 0 of the information block of flash memory).

If the fuse has been blown the Program Mode command is ignored, and the QA Chip returns to *Idle Mode*.

If the fuse is still intact, the chip enters Program Mode and erases the entire contents of Flash memory. The QA Chip then validates the erasure. If the erasure was successful, the QA Chip receives up to 4096 bytes of data corresponding to the new program code and variable data. The bytes are transferred in order $byte_0$ to $byte_{4095}$.

Once all bytes of data have been loaded into Flash, the QA Chip returns to *Idle Mode*.

Note that Trim Mode functionality *must be performed before* a chip enters Program Mode for the first time.

Once the desired number of bytes have been downloaded in Program Mode, the LSS Master must wait for 80µs (the time taken to write two bytes to flash at nybble rates) before sending the new transaction (eg Active Mode). Otherwise the last nybbles may not be written to flash.

15.1.4 Active Mode

Active Mode is entered either by receiving a global id byte (0x00) followed by the Active Mode command byte, or by sending a local id byte followed by a command opcode byte and an appropriate number of data bytes representing the required input parameters for that opcode.

In both cases, Active Mode causes execution of program code previously stored in the flash memory via Program Mode. As a result, we never enter Active Mode after Trim Mode, without a Program Mode in between. However once programmed via Program Mode, a chip is allowed to enter Active Mode after power-up, since valid data will be in flash.

If Active Mode is entered by the global id mechanism, the QA Chip executes specific reset startup code, typically setting up the local id and other IO specific data.

If Active Mode is entered by the local id mechanism, the QA Chip executes specific code depending on the following byte, which functions as an opcode. The opcode command byte format is shown in Table 9:

Table 9. Command byte

Bits	Description
2-0	opcode
5-3	~opcode
7-6	count of number of bits set in opcode (0 to 3)

The interpretation of the 3-bit opcode is shown in Table 10:

Table 10. QA Chip opcodes

Op.[a]	Mnem.[b]	Description
000	RST	Reset
001	RND	Random
010	RDM	Read M
011	TST	Test
100	WRM	Write M with no authentication
101	WRA	Write with Authentication (to M, P, or K)
110	chip specific - reserved for ChipF, ChipS etc	
111	chip specific - reserved for ChipF, ChipS etc	

a. Opcode
b. Mnemonic

The command byte is designed to ensure that errors in transmission are detected.

Regular QA Chip commands are therefore comprised of an opcode plus any associated parameters. The commands are listed in Table 11:

Table 11. QA Chip commands

Command	Input		Output
	Opcode	Additional parms	Return value
Reset	RST	-	-
Random	RND	-	[20]
Read	RDM	[1, 1, 20]	[20, 64, 20][a]
Test	TST	[1, 20, 64, 20]	89[b] if successful, 76 if not
Write	WRM	[1, 64, 20]	89 if successful, 76 if not
WriteAuth	WRA	76 [20, 64, 20]	89 if successful, 76 if not
ReplaceKey	WRA	89 76 [1, 20, 20, 20]	89 if successful, 76 if not
SetPermissions	WRA	89 89 [1, 1, 20, 4, 20]	[4]
SignM[c]	ChipS only	[1, 20, 20, 64, 20, 64]	[20, 64, 20]
SignP[d]	ChipS only	[1, 20, 20, 4, 20, 4]	[20, 64, 20]
GetProgKey	ChipF only	[1, 20]	[20, 20, 20]
SetPartialKey	ChipF only	[1, 4]	89 if successful, 76 if not

a. [n, m] = list of parameters where n bytes for first parameter, and m bytes for the second etc.
b. n = actual byte pattern required (in hex). The bytes 0x76 and 0x89 were chosen as the boolean values 0 and 1 as they are inverses of each other, and should not be generated accidentally.
c. It is expected that most QA Chips will implement SignM as a function that returns 0x00. Only a limited number of chips will be programmed to allow SignM functionality. It is included here as an example of how signatures can be generated for authenticated writes.
d. It is expected that most QA Chips will implement SignP as a function that returns 0x00. Only a limited number of chips will be programmed to allow SignP functionality. It is included here as an example of how signatures can be generated for authenticated writes.

Apart from the Reset command, the next four commands are the commands most likely to be used during regular operation. The next three commands are used to provide authenticated writes (which are expected to be uncommon). The final set of commands (including SignM), are expected to be specially implemented on ChipS and ChipF QA Chips only.

The input parameters are sent in the specified order, with each parameter being sent least significant byte first and most significant byte last.

Return (output) values are read in the same way - least significant byte first and most significant byte last. The client must know how many bytes to retrieve. The QA Chip will time out and return to *Idle Mode* if an incorrect number of bytes is provided or read.

In most cases, the output bytes from one chip's command (the return values) can be fed directly as the input bytes to another chip's command. An example of this is the RND and RD commands. The output data from a call to RND on a trusted QA Chip does not have to be kept by the System. Instead, the System can transfer the output bytes directly to the input of the non-trusted QA Chip's RD command. The description of each command points out where this is so.

Each of the commands is examined in detail in the subsequent sections. Note that some algorithms are specifically designed because flash memory is assumed for the implementation of non-volatile variables.

15.1.5 Non volatile variables

The memory within the QA Chip contains some *non-volatile* (Flash) memory to store the variables required by the authentication protocol. Table 12 summarizes the variables.

Table 12. Non volatile variables required by the authentication protocol

Name	Size (bits)	Description
N	8	Number of keys known to the chip
T	8	Number of vectors M is broken into
K_n R_K	160 per key, 160 for R_K	Array of *N* secret keys used for calculating $F_{Kn}[X]$ where K_n is the *n*th element of the array. Each K_n must not be stored directly in the QA Chip. Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored $K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.
R	160	Current random number used to ensure time varying messages. Each chip instance must be seeded with a different initial value. Changes for each signature generation.
M_T	512 per M	Array of T memory vectors. Only M_0 can be written to with an authorized write, while all Ms can be written to in an unauthorized write. Writes to M_0 are optimized for Flash usage, while updates to any other M_n are expensive with regards to Flash utilization, and are expected to be only performed once per section of M_n. M_1 contains T and N in ReadOnly form so users of the chip can know these two values.
P_{T+N}	32 per P	T+N element array of access permissions for each part of M. Entries n={0... T-1} hold access permissions for non-authenticated writes to M_n (no key required). Entries n={T to T+N-1}hold access permissions for authenticated writes to M_0 for K_n. Permission choices for each part of M are Read Only, Read/Write, and Decrement Only
MinTicks	32	The minimum number of clock ticks between calls to key-based functions.

Note that since these variables are in Flash memory, writes should be minimized. The it is not a simple matter to write a new value to replace the old. Care must be taken with flash endurance, and speed of access. This has an effect on the algorithms used to change Flash memory based registers. For example, Flash memory should not be used as a shift register.

A reset of the QA Chip has no effect on the non-volatile variables.

15.1.5.1 M and P

M_n contains application specific state data, such as serial numbers, batch numbers, and amount of consumable remaining. M_n can be read using the Read command and written to via the Write and WriteA commands.

M_0 is expected to be updated frequently, while each part of M_{1-n} should only be written to once. Only M_0 can be written to via the WriteA command.

M_1 contains the operating parameters of the chip as shown in Table 13, and M_{2-n} are application specific.

Table 13. Interpretation of M_1

Length	Bits	Interpretation
8	7-0	Number of available keys
8	15-8	Number of available M vectors
16	31-16	Revision of chip
96	127-32	Manufacture id information
128	255-128	Serial number
8	263-256	Local id of chip
248	511-264	reserved

Each M_n is 512 bits in length, and is interpreted as a set of 16×32-bit words. Although M_n may contain a number of different elements, each 32-bit word differs only in write permissions. Each 32-bit word can always be read. Once in client memory, the 512 bits can be interpreted in any way chosen by the client. The different write permissions for each P are outlined in Table 14:

Table 14. Write permissions

Data type	permission description
Read Only	Can *never* be written to
ReadWrite	Can *always* be written to
Decrement Only	Can only be written to if the new value is less than the old value. Decrement Only values can be any multiple of 32 bits.

To accomplish the protection required for writing, a 2-bit permission value P is defined for each of the 32-bit words. Table 15 defines the interpretation of the 2-bit permission bit-pattern:

Table 15. Permission bit interpretation

Bits	Op	Interpretation	Action taken during Write command
00	RW	ReadWrite	The new 32-bit value is always written to M[n].
01	MSR	Decrement Only (Most Significant Region)	The new 32-bit value is only written to M[n] if it is less than the value currently in M[n]. This is used for access to the Most Significant 16 bits of a Decrement Only number.
10	NMSR	Decrement Only (Not the Most Significant Region)	The new 32-bit value is only written to M[n] if M[n-1] could also be written. The NMSR access mode allows multiple precision values of 32 bits and more (multiples of 32 bits) to decrement.
11	RO	Read Only	The new 32-bit value is ignored. M[n] is left unchanged.

The 16 sets of permission bits for each 512 bits of M are gathered together in a single 32-bit variable P, where bits 2n and 2n+1 of P correspond to word n of M as follows:

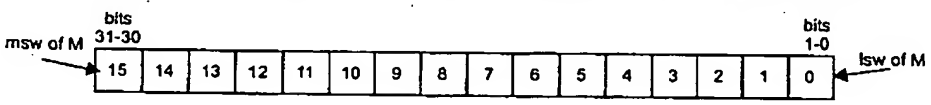


Figure 13. Relationship of Permissions bits to M[n] access bits

Each 2-bit value is stored as a pair with the msb in bit 1, and the lsb in bit 0. Consequently, if words 0 to 5 of M had permission MSR, with words 6-15 of M permission RO, the 32-bit P variable would be `0xFFFFF555`:

11-11-11-11-11-11-11-11-11-11-01-01-01-01-01-01

During execution of a Write and WriteA command, the appropriate Permissions[n] is examined for each M[n] starting from n=15 (msw of M) to n=0 (lsw of M), and a decision made as to whether the new M[n] value will replace the old. Note that it is important to process the M[n] from msw to lsw to correctly interpret the access permissions.

Permissions are set and read using the QA Chip's SetPermissions command. The default for P is all 0s (RW) with the exception of certain parts of M_1.

Note that the Decrement Only comparison is *unsigned*, so any Decrement Only values that require negative ranges must be shifted into a positive range. For example, a consumable with a Decrement Only data item range of -50 to 50 must have the range shifted to be 0 to 100. The System must then interpret the range 0 to 100 as being -50 to 50. Note that most instances of Decrement Only ranges are N to 0, so there is no range shift required.

For Decrement Only data items, arrange the data in order *from most significant to least significant* 32-bit quantities from M[n] onward. The access mode for the most significant 32 bits (stored in M[n]) should be set to MSR. The remaining 32-bit entries for the data should have their permissions set to NMSR.

If erroneously set to NMSR; with no associated MSR region, each NMSR region will be considered independently instead of being a multi-precision comparison.

Examples of allocating M and Permission bits can be found in [86].

15.1.5.2 K and R_K

K is the 160-bit secret key used to protect M and to ensure that the contents of M are valid (when M is read from a non trusted chip). K is initially programmed after manufacture, and from that point on, K can only be updated to a new value if the old K is known. Since K must be kept secret, there is no command to directly read it.

K is used in the keyed one-way hash function HMAC-SHA1. As such it should be programmed with a *physically generated* random number, gathered from a physically random phenomenon. ***K must NOT be generated with a computer-run random number generator.*** The security of the QA Chips depends on K being generated in a way that is not deterministic.

Each K_n must not be stored directly in the QA Chip. Instead, each chip needs to store a single random number R_K (different for each chip), $K_n \oplus R_K$, and $\neg K_n \oplus R_K$. The stored

$K_n \oplus R_K$ can be XORed with R_K to obtain the real K_n. Although $\neg K_n \oplus R_K$ must be stored to protect against differential attacks, it is not used.

15.1.5.3 R

R is a 160-bit random number seed that is set up after manufacture (when the chip is programmed) and from that point on, cannot be changed. R is used to ensure that each signed item contains time varying information (not chosen by an attacker), and each chip's R is unrelated from one chip to the next.

R is used during the Test command to ensure that the R from the previous call to Random was used as the session key in generating the signature during Read. Likewise, R is used during the WriteAuth command to ensure that the R from the previous call to Read was used as the session key during generation of the signature in the remote Authenticated chip.

The only invalid value for R is 0. This is because R is changed via a 160-bit maximal period LFSR (Linear Feedback Shift Register) with taps on bits 0, 2, 3, and 5, and is changed only by a successful call to a signature generating function (e.g. Test, WriteAuth).

The logical security of the QA Chip relies not only upon the randomness of K and the strength of the HMAC-SHA1 algorithm. To prevent an attacker from building a sparse lookup table, the security of the QA Chip also depends on the range of R over the lifetime of *all* Systems. What this means is that an attacker must not be able to deduce what values of R there are in produced and future Systems. Ideally, R should be programmed with a *physically generated* random number, gathered from a physically random phenomenon (must not be deterministic). ***R must NOT be generated with a computer-run random number generator.***

15.1.5.4 MinTicks

There are two mechanisms for preventing an attacker from generating multiple calls to key-based functions in a short period of time. The first is an internal ring oscillator that is temperature-filtered. The second mechanism is the 32-bit MinTicks variable, which is used to specify the minimum number of QA Chip clock ticks that must elapse between calls to key-based functions.

The MinTicks variable is set to a fixed value when the QA Chip is programmed. It could possibly be stored in M_1.

The effective value of MinTicks depends on the *operating* clock speed and the notion of what constitutes a reasonable time between key-based function calls (application specific). The duration of a single tick depends on the operating clock speed. This is the fastest speed of the ring oscillator generated clock (i.e. at the lowest valid operating temperature).

Once the duration of a tick is known, the MinTicks value can to be set. The value for MinTicks will be the minimum number of ticks required to pass between calls to the key-based functions (there is no need to protect Random as this produces the same output each time it is called multiple times in a row). The value is a real-time number, and divided by the length of an operating tick.

It should be noted that the MinTicks variable *only slows down* an attacker and causes the attack to cost more since it does not stop an attacker using multiple System chips in parallel.

15.1.6 GetProgramKey

Input: n, R_E = [1 byte, 20 bytes]

Output: R_L, $E_{Kx}[S_{Kn}[R_E|R_L|C_3]]$, $S_{Kx}[R_L|E_{Kx}[S_{Kn}[R_E|R_L|C_3]|C_3]$ = [20, 20, 20]

Changes: R_L

*Note: The **GetProgramKey** command is only implemented in ChipF, and not in all QA Chips.*

The ***GetProgramKey*** command is used to produce the bytestream required for updating a specified key in ChipP. Only an QA Chip programmed with the correct values of the old K_n can respond correctly to the GetProgramKey request. The output bytestream from the Random command can be fed as the input bytestream to the ReplaceKey command on the QA Chip being programmed (ChipP).

The input bytestream consists of the appropriate opcode followed by the desired key to generate the signature, followed by 20 bytes of R_E(representing the random number read in from ChipP).

The local random number R_L is advanced, and signed in combination with R_E and C_3 by the chosen key to generate a time varying secret number known to both ChipF and ChipP. This signature is then XORed with the new key K_x (this encrypts the new key). The first two output parameters are signed with the old key to ensure that ChipP knows it decoded K_x correctly.

This whole procedure should only be allowed a given number of times. The actual number can conveniently be stored in the local $M_0[0]$ (eg word 0 of M_0) with ReadOnly permission. Of course another chip could perform an Authorised write to update the number (via a ChipS) should it be desired.

The GetProgramKey command is implemented by the following steps:

```
Loop through all of Flash, reading each word (will trigger checks)
Accept n
Restrict n to N
Accept R_E
If (M_0[0] = 0)
   Output 60 bytes of 0x00# no more keys allowed to be generated from this chipF
   Done
EndIf

Advance R_L
SIG ← S_Kn[R_L|R_E|C_3]# calculation must take constant time
Tmp ← SIG ⊕ K_X
Output R_L
Output Tmp
Decrement M_0[0]# reduce the number of allowable key generations by 1
SIG ← S_KX[R_L|Tmp|C_3]# calculation must take constant time
Output SIG
```

15.1.7 Random

Input: None

Output: R_L = [20 bytes]

Changes: None

The ***Random*** command is used by a client to obtain an input for use in a subsequent authentication procedure. Since the Random command requires no input parameters, it is therefore simply 1 byte containing the RND opcode.

The output of the Random command from a trusted QA Chip can be fed straight into the non-trusted chip's Read command as part of the input parameters. There is no need for the client to store them at all, since they are not required again. However the Test command will only succeed if the data passed to the Read command was obtained first from the Random command.

If a caller only calls the Random function multiple times, the same output will be returned each time. R will only advance to the next random number in the sequence after a successful call to a function that returns or tests a signature (e.g. Test, see Section 15.1.13 on page 91 for more information).

The Random command is implemented by the following steps:

```
Loop through all of Flash, reading each word (will trigger checks)
Output R_L
```

15.1.8 Read

Input: n, t, R_E = [1 byte, 1 byte, 20 bytes]

Output: R_L, M_{Lt}, $S_{Kn}[R_E|R_L|C_1|M_{Lt}]$ = [20 bytes, 64 bytes, 20 bytes]

Changes: R_L

The ***Read*** command is used to read the entire state data (M_t) from an QA Chip. Only an QA Chip programmed with the correct value of K_n can respond correctly to the Read request. The output bytestream from the Read command can be fed as the input bytestream to the Test command on a trusted QA Chip for verification, with M_t stored for later use if Test returns success.

The input bytestream consists of the RD opcode followed by the key number to use for the signature, which M to read, and the bytes 0-19 of R_E. 23 bytes are transferred in total. R_E is obtained by calling the trusted QA Chip's Random command. The 20 bytes output by the trusted chip's Random command can therefore be fed directly into the non-trusted chip's Read command, with no need for these bits to be stored by System.

Calls to Read must wait for MinTicksRemaining to reach 0 to ensure that a minimum time will elapse between calls to Read.

The output values are calculated, MinTicksRemaining is updated, and the signature is returned. The contents of M_{Lt} are transferred least significant byte to most significant byte. The signature $S_{Kn}[R_E|R_L|C_1|M_{Lt}]$ must be calculated in constant time.

The next random number is generated from R using a 160-bit maximal period LFSR (tap selections on bits 5, 3, 2, and 0). The initial 160-bit value for R is set up when the chip is programmed, and can be any random number except 0 (an LFSR filled with 0s will produce a never-ending stream of 0s). R is transformed by XORing bits 0, 2, 3, and 5 together, and shifting all 160 bits right 1 bit using the XOR result as the input bit to b_{159}. The process is shown in Figure 14:.

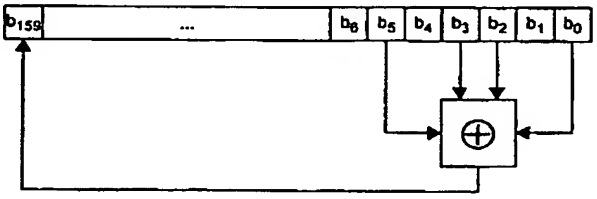


Figure 14. 160 bit maximal period LFSR

Care should be taken when updating R since it lives in Flash. Program code must assume power could be removed at any time.

The Read command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)
Accept n
Accept t
Restrict n to N
Restrict t to T
Accept RE
Advance RL
Output RL
```

```
Output M_Lt
Sig ← S_Kn{R_E|R_L|C_1|M_Lt}# calculation must take constant time
MinTicksRemaining ← MinTicks
Output Sig
Wait for MinTicksRemaining to become 0
```

15.1.9 Set Permissions

Input: n, p, R_E, P_E, SIG_E = [1 byte, 1 byte, 20 bytes, 4 bytes, 20 bytes]

Output: P_p

Changes: P_p, R_L

The ***SetPermissions*** command is used to securely update the contents of P_p (containing QA Chip permissions). The WriteAuth command only attempts to replace P_p if the new value is signed combined with our local R.

It is only possible to sign messages by knowing K_n. This can be achieved by a call to the SignP command (because only a ChipS can know K_n). It means that without a chip that can be used to produce the required signature, a write of any value to P_p is not possible.

The process is very similar to Test, except that if the validation succeeds, the P_E input parameter is additionally ORed with the current value for P_p. Note that this is an OR, and not a replace. Since the SetParms command only sets bits in P_p, the effect is to allow the permission bits corresponding to M[n] to progress from RW to either MSR, NMSR, or RO.

The SetPermissions command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)

Accept n
Restrict n to N
Accept p
Restrict p to T+N
Accept R_E
Accept P_E
SIG_L ← S_Kn{R_L|R_E|P_E|C_2}# calculation must take constant time
Accept SIG_E
If (SIG_E = SIG_L)
   Update R_L
   P_P ← P_P v P_E
EndIf
Output P_P# success or failure will be determined by receiver
MinTicksRemaining ← MinTicks
```

15.1.10 ReplaceKey

Input: n, R_E, V, SIG_E = [1 byte, 20 bytes, 20 bytes, 20 bytes]

Output: Boolean (0x76=failure, 0x89 = success)

Changes: K_n, M_L, R_L

The ***ReplaceKey*** command is used to replace the specified key in the QA Chip flash memory. However K_n can only be replaced if the previous value is known. A return byte of 0x89 is produced if the key was successfully updated, while 0x76 is returned for failure.

A ReplaceKey command consists of the WRA command opcode followed by 0x89, 0x76, and then the appropriate parameters. Note that the new key is not sent in the clear, it is sent encrypted with the signature of R_L, R_E and C_3 (signed with the old key). The first two input parameters must be verified by generating a signature using the old key.

The ReplaceKey command is implemented with the following steps:

```
Loop through all of Flash, reading each word (will trigger checks)
Accept n
Restrict n to N
Accept RE# session key from ChipF
Accept V # encrypted key

SIGL ← SKn[RE|V|C3]# calculation must take constant time
Accept SIGE
If (SIGL = SIGE2) # comparison must take constant time
   SIGL ← SKn[RL|RE|C3]# calculation must take constant time
   Advance RL
   KE ← SIGL ⊕ V
   Kn ← KE           # involves storing (KE ⊕ RE) and (¬KE ⊕ RE)
   Output 0x89# success
Else
   Output 0x76# failure
EndIf
```

15.1.11 SignM

Input: $n, R_X, R_E, M_E, SIG_E, M_{desired}$ = [1 byte, 20 bytes, 20 bytes, 64 bytes, 32 bytes]

Output: R_L, M_{new}, $S_{Kn}[R_E \mid R_L \mid C_1 \mid M_{new}]$ = [20 bytes, 64 bytes, 20 bytes]

Changes: R_L

Note: The SignM command is only implemented in ChipS, and not in all QA Chips.

The ***SignM*** command is used to produce a valid signed M for use in an authenticated write transaction. Only an QA Chip programmed with correct value of K_n can respond correctly to the SignM request. The output bytestream from the SignM command can be fed as the input bytestream to the WriteA command on a different QA Chip.

The input bytestream consists of the SMR opcode followed by 1 byte containing the key number to use for generating the signature, 20 bytes of R_X (representing the number passed in as R to ChipU's READ command, i.e. typically 0), the output from the READ command (namely R_E, M_E, and SIG_E), and finally the desired M to write to ChipU.

The SignM command only succeeds when $SIG_E = S_K[R_X \mid R_E \mid C_1 \mid M_E]$, indicating that the request was generated from a chip that knows K. This generation and comparison *must take the same amount of time regardless of whether the input parameters are correct or not.* If the times are not the same, an attacker can gain information about which bits of the supplied signature are incorrect. If the signatures match, then R_L is updated to be the next random number in the sequence.

Since the SignM function generates signatures, the function must wait for the MinTicksRemaining register to reach 0 before processing takes place.

Once all the inputs have been verified, a new memory vector is produced by applying a specially stored P value (eg word 1 of M_0) and $M_{desired}$ against M_E. Effectively, it is performing a regular Write, but with separate P against someone else's M. The M_{new} is signed with an updated R_L (and the passed in R_E), and all three values are output (the random number R_L, M_{new} and the signature). The time taken to generate this signature must be the same regardless of the inputs.

Typically, the SignM command will be acting as a form of consumable command, so that a given ChipS can only generate a given number of signatures. The actual number can conveniently be stored in M_0 (eg word 0 of M_0) with ReadOnly permissions. Of course another chip could perform an Authorised write to update the number (using another ChipS) should it be desired.

The SignM command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)

Accept n
Restrict n to N
Accept R_X                  # don't care what this number is
Accept R_E
Accept M_E
SIG_L ← S_Kn[R_X|R_E|C_1|M_E]# calculation must take constant time
Accept SIG_E
Accept M_desired
If ((SIG_E ≠ SIG_L) OR (M_L[0] = 0))# fail if bad signature or if allowed sigs = 0
   Output appropriate number of 0# report failure
```

```
   Done
EndIf

Update R_L

# Create the new version of M in ram from W and Permissions
# This is the same as the core process of Write function
# except that we don't write the results back to M
DecEncountered ← 0
EqEncountered ← 0
Permissions = M_L[1]          # assuming M_0 contains appropriate permissions
For n ← msw to lsw #(word 15 to 0)
   AM ← Permissions[n]
   LT ← (M_desired[n] < M_E[n])# comparison is unsigned
   EQ ← (M_desired[n] = M_E[n])
   WE ← (AM = RW) ∨ ((AM = MSR) ∧ LT) ∨ ((AM = NMSR) ∧ (DecEncountered ∨ LT))
   DecEncountered ← ((AM = MSR) ∧ LT)
                  ∨ ((AM = NMSR) ∧ DecEncountered)
                  ∨ ((AM = NMSR) ∧ EqEncountered ∧ LT)
   EqEncountered ← ((AM = MSR) ∧ EQ) ∨ ((AM = NMSR) ∧ EqEncountered ∧ EQ)
   If (¬WE) ∧ (M_E[n] ≠ M_desired[n])
      Output appropriate number of 0# report failure
   EndIf
EndFor

# At this point, M_desired is correct
Output R_L
Output M_desired      # M_desired is now effectively M_new
Sig ← S_Kn[R_E|R_L|C_1|M_desired]# calculation must take constant time
MinTicksRemaining ← MinTicks
Decrement M_L[0]# reduce the number of allowable signatures by 1
Output Sig
```

15.1.12 SignP

Input: $n, R_E, P_{desired}$ = [1 byte, 20 bytes, 4 bytes]

Output: R_L, $S_{Kn}[R_E | R_L | P_{desired} | C_2]$ = [20 bytes, 20 bytes]

Changes: R_L

Note: The SignP command is only implemented in ChipS, and not in all QA Chips.

The ***SignP*** command is used to produce a valid signed P for use in a SetPermissions transaction. Only an QA Chip programmed with correct value of K_n can respond correctly to the SignP request. The output bytestream from the SignP command can be fed as the input bytestream to the SetPermissions command on a different QA Chip.

The input bytestream consists of the SMP opcode followed by 1 byte containing the key number to use for generating the signature, 20 bytes of R_E (representing the number obtained from ChipU's RND command, and finally the desired P to write to ChipU.

Since the SignP function generates signatures, the function must wait for the MinTicksRemaining register to reach 0 before processing takes place.

Once all the inputs have been verified, the $P_{desired}$ is signed with an updated R_L (and the passed in R_E), and both values are output (the random number R_L and the signature). The time taken to generate this signature must be the same regardless of the inputs.

Typically, the SignP command will be acting as a form of consumable command, so that a given ChipS can only generate a given number of signatures. The actual number can conveniently be stored in M_0 (eg word 0 of M_0) with ReadOnly permissions. Of course another chip could perform an Authorised write to update the number (using another ChipS) should it be desired.

The SignM command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)

Accept n
Restrict n to N
Accept R_E
Accept P_desired
If (M_L[0] = 0)# fail if allowed sigs = 0
   Output appropriate number of 0# report failure
   Done
EndIf

Update R_L
Output R_L
Sig ← S_Kn[R_E|R_L|P_desired|C_2]# calculation must take constant time
MinTicksRemaining ← MinTicks
Decrement M_L[0]# reduce the number of allowable signatures by 1
Output Sig
```

15.1.13 Test

Input: n, R_E, M_E, SIG_E = [1 byte, 20 bytes, 64 bytes, 20 bytes]

Output: Boolean (0x76=failure, 0x89 = success)

Changes: R_L

The *Test* command is used to authenticate a read of an M from a non-trusted QA Chip.

The Test command consists of the TST command opcode followed by input parameters: n, R_E, M_E, and SIG_E. The byte order is least significant byte to most significant byte for each command component. All but the first input parameter bytes are obtained as the output bytes from a Read command to a non-trusted QA Chip. The entire data does not have to be stored by the client. Instead, the bytes can be passed directly to the trusted QA Chip's Test command, and only M should be kept from the Read.

Calls to Test must wait for the MinTicksRemaining register to reach 0.

$S_{Kn}[R_L|R_E|C_1|M_E]$ is then calculated, and compared against the input signature SIG_E. If they are different, R_L is not changed, and 0x76 is returned to indicate failure. If they are the same, then R_L is updated to be the next random number in the sequence and 0x89 is returned to indicate success. Updating R_L only after success forces the caller to use a new random number (via the Random command) each time a successful authentication is performed.

The calculation of $S_{Kn}[R_L|R_E|C_1|M_E]$ and the comparison against SIG_E must take identical time so that *the time to evaluate the comparison in the TST function is always the same.* Thus no attacker can compare execution times or number of bits processed before an output is given.

The Test command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)

Accept n
Restrict n to N
Accept R_E
Accept M_E
SIG_L ← S_Kn[R_L|R_E|C_1|M_E]# calculation must take constant time
Accept SIG_E
If (SIG_E = SIG_L)
   Update R_L
   Output 0x89# success
Else
   Output 0x76       # report failure
EndIf
MinTicksRemaining ← MinTicks
```

15.1.14 Write

Input: t, M_{new}, SIG_E = [1 byte, 64 bytes, 20 bytes]

Output: Boolean (0x76=failure, 0x89 = success)

Changes: M_t

The *Write* command is used to update M_t according to the permissions in P_t. *The WR command by itself is not secure, since a clone QA Chip may simply return success every time.* Therefore a Write command should be followed by an authenticated read of M_t (e.g. via a Read command) to ensure that the change was actually made.

The Write command is called by passing the WR command opcode followed by which M to be updated, the new data to be written to M, and a digital signature of M. The data is sent least significant byte to most significant byte.

The ability to write to a specific 32-bit word within M_t is governed by the corresponding Permissions bits as stored in P_t. P_t can be set using the SetPermissions command.

The fact that M_t is Flash memory must be taken into account when writing the new value to M. It is possible for an attacker to remove power at any time. In addition, only the changes to M should be stored for maximum utilization. In addition, the longevity of M will need to be taken into account. This may result in the location of M being updated.

The signature is not keyed, since it must be generated by the consumable user.

The Write command is implemented with the following steps:

```
Loop through all of Flash, reading each word (will trigger checks)
Accept t
Restrict t to T
Accept Mg# new M
Accept SIGg

SIGL = Generate SHA1[Mg]
If (SIGL = SIGg)
   output 0x76# failure due to invalid signature
   exit
EndIf
DecEncountered ← 0
EqEncountered ← 0
For i ← msw to lsw #(word 15 to 0)
   P ← Pt[i]
   LT ← (Mg[i] < Mt[i])# comparison is unsigned
   EQ ← (Mg[i] = Mt[i])
   WE ← (P = RW) ∨ ((P = MSR) ∧ LT) ∨ ((P = NMSR) ∧ (DecEncountered ∨ LT))
   DecEncountered ← ((P = MSR)' ∧ LT)
                  ∨ ((P = NMSR) ∧ DecEncountered)
                  ∨ ((P = NMSR) ∧ EqEncountered ∧ LT)
   EqEncountered ← ((P = MSR) ∧ EQ) ∨ ((P = NMSR) ∧ EqEncountered ∧ EQ)

   If (¬WE) ∧ (Mg[i] ≠ Mt[i])
      output 0x76# failure due to wanting a change but not allowed it
   EndIf
EndFor

# At this point, Mg (desired) is correct to be written to the flash
Mt ← Mg        # update flash
output 0x89    # success
```

15.1.15 WriteAuth

Input: n, R_E, M_E, SIG_E = [1 byte, 20 bytes, 64 bytes, 20 bytes]

Output: Boolean (0x76=failure, 0x89 = success)

Changes: M_0, R_L

The ***WriteAuth*** command is used to securely replace the entire contents of M_0 (containing QA Chip application specific data) according to the P_{T+n}. The WriteAuth command only attempts to replace M_0 if the new value is signed combined with our local R.

It is only possible to sign messages by knowing K_n. This can be achieved by a call to the SignM command (because only a ChipS can know K_n). It means that without a chip that can be used to produce the required signature, a write of any value to M_0 is not possible.

The process is very similar to Write, except that if the validation succeeds, the M_E input parameter is processed against M_0 using permissions P_{T+n}.

The WriteAuth command is implemented with the following steps:

```
Wait for MinTicksRemaining to become 0
Loop through all of Flash, reading each word (will trigger checks)

Accept n
Restrict n to N
Accept R_E
Accept M_E
SIG_L ← S_Kn[R_L|R_E|C_1|M_E]# calculation must take constant time
Accept SIG_E
If (SIG_E = SIG_L)
   Update R_L
   DecEncountered ← 0
   EqEncountered ← 0
   For i ← msw to lsw #(word 15 to 0)
      P ← P_T+n[i]
      LT ← (M_E[i] < M_0[i])# comparison is unsigned
      EQ ← (M_E[i] = M_0[i])
      WE ← (P = RW) ∨ ((P = MSR) ∧ LT) ∨ ((P = NMSR) ∧ (DecEncountered ∨ LT))
      DecEncountered ← ((P = MSR) ∧ LT)
                     ∨ ((P = NMSR) ∧ DecEncountered)
                     ∨ ((P = NMSR) ∧ EqEncountered ∧ LT)
      EqEncountered ← ((P = MSR) ∧ EQ) ∨ ((P = NMSR) ∧ EqEncountered ∧ EQ)
      If ((¬WE) ∧ (M_E[i] ≠ M_0[i]))
         output 0x76# failure due to wanting a change but not allowed it
      EndIf
   EndFor
   # At this point, M_E (desired) is correct to be written to the flash
   M_0 ← M_E      # update flash
   output 0x89    # success
EndIf
MinTicksRemaining ← MinTicks
```

16 Manufacture

This chapter makes some general comments about the manufacture and implementation of authentication chips. While the comments presented here are general, see [84] for a detailed description of an implementation of an authentication chip.

The authentication chip algorithms do not constitute a strong encryption device. The net effect is that they can be safely manufactured in any country (including the USA) and exported to anywhere in the world.

The circuitry of the authentication chip must be resistant to physical attack. A summary of manufacturing implementation guidelines is presented, followed by specification of the chip's physical defenses (ordered by attack).

Note that manufacturing comments are in addition to any legal protection undertaken, such as patents, copyright, and license agreements (for example, penalties if caught reverse engineering the authentication chip).

16.1 Guidelines for Manufacturing

The following are general guidelines for implementation of an authentication chip in terms of manufacture (see [84] for a detailed description of an authentication chip). *No special security is required during the manufacturing process.*

- Standard process
- Minimum size (if possible)
- Clock Filter
- Noise Generator
- Tamper Prevention and Detection circuitry
- Protected memory with tamper detection
- Boot circuitry for loading program code
- Special implementation of FETs for key data paths
- Data connections in polysilicon layers where possible
- OverUnderPower Detection Unit
- No test circuitry
- Transparent epoxy packaging

Finally, as a general note to manufacturers of Systems, the data line to the System authentication chip and the data line to the Consumable authentication chip must not be the same line. See Section 16.2.3 on page 103.

16.1.1 Standard Process

The authentication chip should be implemented with a standard manufacturing process (such as Flash). This is necessary to:

- allow a great range of manufacturing location options
- take advantage of well-defined and well-behaved technology
- reduce cost

Note that the standard process still allows physical protection mechanisms.

16.1.2 Minimum size

The authentication chip must have a low manufacturing cost in order to be included as the authentication mechanism for low cost consumables. It is therefore desirable to keep the chip size as low as reasonably possible.

Each authentication chip requires 962 bits of non-volatile memory. In addition, the storage required for optimized HMAC-SHA1 is 1024 bits. The remainder of the chip (state machine, processor, CPU or whatever is chosen to implement Protocol C1) must be kept to a minimum in order that the number of transistors is minimized and thus the cost per chip is minimized. The circuit areas that process the secret key information or could reveal information about the key should also be minimized (see Section 16.1.8 on page 100 for special data paths).

16.1.3 Clock Filter

The authentication chip circuitry is designed to operate within a specific clock speed range. Since the user directly supplies the clock signal, it is possible for an attacker to attempt to introduce race-conditions in the circuitry at specific times during processing. An example of this is where a high clock speed (higher than the circuitry is designed for) may prevent an XOR from working properly, and of the two inputs, the first may always be returned. These styles of transient fault attacks can be very efficient at recovering secret key information, and have been documented in [5] and [1]. The lesson to be learned from this is that the input clock signal *cannot be trusted*.

Since the input clock signal cannot be trusted, it must be limited to operate up to a maximum frequency. This can be achieved a number of ways.

One way to filter the clock signal is to use an edge detect unit passing the edge on to a delay, which in turn enables the input clock signal to pass through.

Figure 15 shows clock signal flow within the Clock Filter:

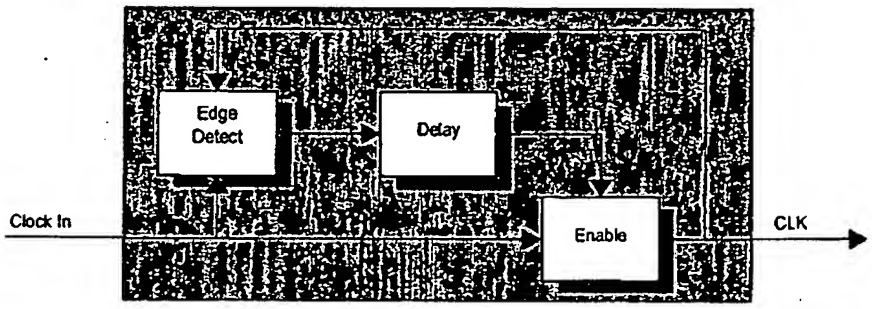


Figure 15. Clock Filter

The delay should be set so that the maximum clock speed is a particular frequency (e.g. about 4 MHz). Note that this delay is not programmable - it is fixed.

The filtered clock signal would be further divided internally as required.

16.1.4 Noise Generator

Each authentication chip should contain a noise generator that generates continuous circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and add noise to the I_{dd} signal. Placement of the noise generator is not an issue on an authentication chip due to the length of the emission wavelengths.

The noise generator is used to generate electronic noise, multiple state changes each clock cycle, and as a source of pseudo-random bits for the Tamper Prevention and Detection circuitry (see Section 16.1.5 on page 96).

A simple implementation of a noise generator is a 64-bit maximal period LFSR seeded with a non-zero number. The clock used for the noise generator should be running at the maximum clock rate for the chip in order to generate as much noise as possible.

16.1.5 Tamper Prevention and Detection circuitry

A set of circuits is required to test for and prevent physical attacks on the authentication chip. However what is actually detected as an attack may not be an intentional physical attack. It is therefore important to distinguish between these two types of attacks in an authentication chip:

- where you *can be certain* that a physical attack has occurred.
- where you *cannot* be certain that a physical attack has occurred.

The two types of detection differ in what is performed as a result of the detection. In the first case, where the circuitry can be certain that a true physical attack has occurred, erasure of Flash memory key information is a sensible action. In the second case, where the circuitry cannot be sure if an attack has occurred, there is still certainly something wrong. Action must be taken, but the action should not be the erasure of secret key information. A suitable action to take in the second case is a chip RESET. If what was detected was an attack that has permanently damaged the chip, the same conditions will occur next time and the chip will RESET again. If, on the other hand, what was detected was part of the normal operating environment of the chip, a RESET will not harm the key.

A good example of an event that circuitry cannot have knowledge about, is a power glitch. The glitch may be an intentional attack, attempting to reveal information about the key. It may, however, be the result of a faulty connection, or simply the start of a power-down sequence. It is therefore best to only RESET the chip, and not erase the key. If the chip was powering down, nothing is lost. If the System is faulty, repeated RESETs will cause the consumer to get the System repaired. In both cases the consumable is still intact.

A good example of an event that circuitry can have knowledge about, is the cutting of a data line within the chip. If this attack is somehow detected, it could only be a result of a faulty chip (manufacturing defect) or an attack. In either case, the erasure of the secret information is a sensible step to take.

Consequently each authentication chip should have 2 Tamper Detection Lines - one for definite attacks, and one for possible attacks. Connected to these Tamper Detection Lines would be a number of Tamper Detection test units, each testing for different forms of tampering. *In addition, we want to ensure that the Tamper Detection Lines and Circuits themselves cannot also be tampered with.*

At one end of the Tamper Detection Line is a source of pseudo-random bits (clocking at high speed compared to the general operating circuitry). The Noise Generator circuit described above is an adequate source. The generated bits pass through two different paths

- one carries the original data, and the other carries the inverse of the data. The wires carrying these bits are in the layer above the general chip circuitry (for example, the memory, the key manipulation circuitry etc.). The wires must also cover the random bit generator. The bits are recombined at a number of places via an XOR gate. If the bits are different (they should be), a 1 is output, and used by the particular unit (for example, each output bit from a memory read should be ANDed with this bit value). The lines finally come together at the Flash memory Erase circuit, where a complete erasure is triggered by a 0 from the XOR. Attached to the line is a number of triggers, each detecting a physical attack on the chip. Each trigger has an oversize nMOS transistor attached to GND. The Tamper Detection Line physically goes through this nMOS transistor. If the test fails, the trigger causes the Tamper Detect Line to become 0. The XOR test will therefore fail on either this clock cycle or the next one (on average), thus RESETing or erasing the chip.

Figure 16 illustrates the basic principle of a Tamper Detection Line in terms of tests and the XOR connected to either the Erase or RESET circuitry.

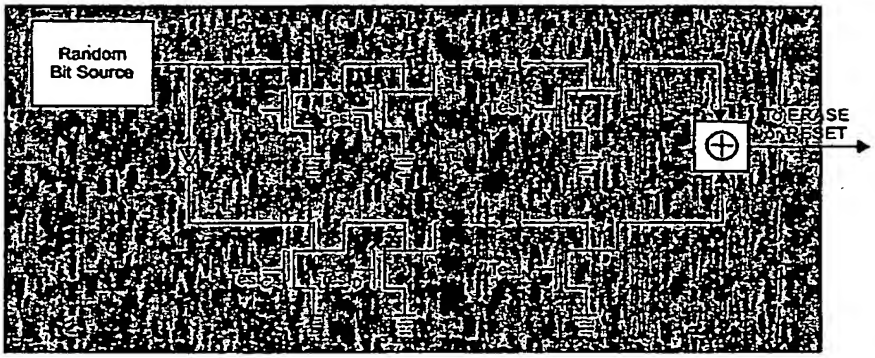


Figure 16. Tamper Detection Line

The Tamper Detection Line must go through the drain of an output transistor for each test, as illustrated by Figure 17:

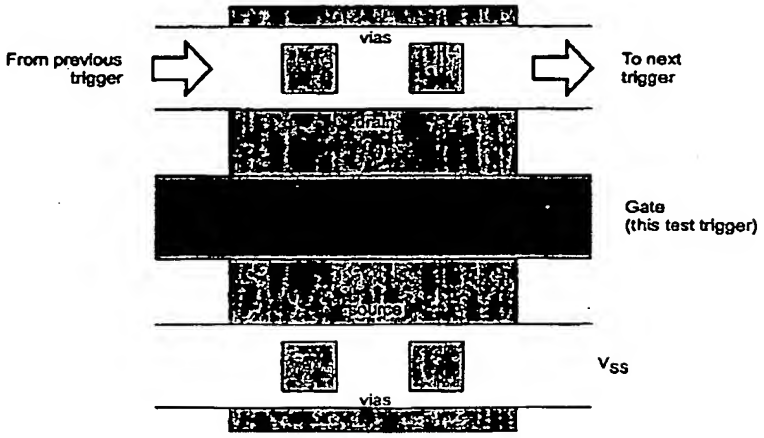


Figure 17. Oversize nMOS Transistor Layout of Tamper Detection Line

It is not possible to break the Tamper Detect Line since this would stop the flow of 1s and 0s from the random source. The XOR tests would therefore fail. As the Tamper Detect Line physically passes through each test, it is not possible to eliminate any particular test without breaking the Tamper Detect Line.

It is important that the XORs take values from a variety of places along the Tamper Detect Lines in order to reduce the chances of an attack. Figure 18 illustrates the taking of multiple XORs from the Tamper Detect Line to be used in the different parts of the chip. Each of these XORs can be considered to be generating a ChipOK bit that can be used within each unit or sub-unit.

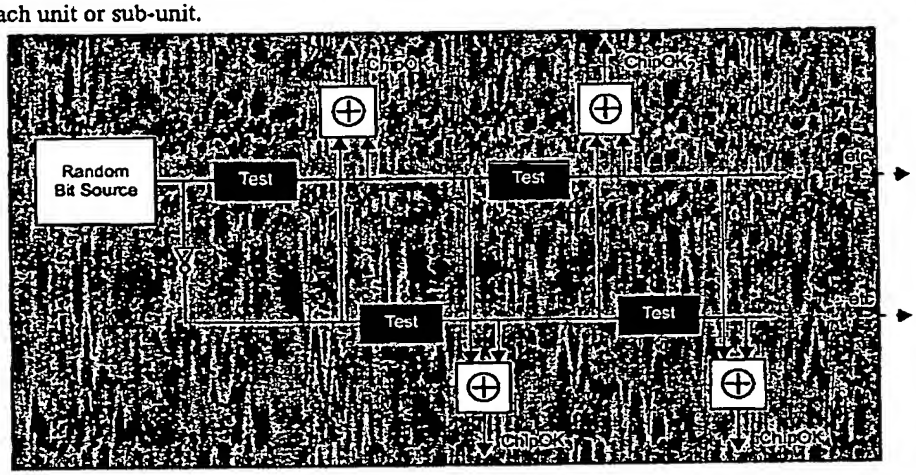


Figure 18. Tamper Detection Line

A sample usage would be to have an OK bit in each unit that is ANDed with a given ChipOK bit each cycle. The OK bit is loaded with 1 on a RESET. If OK is 0, that unit will fail until the next RESET. If the Tamper Detect Line is functioning correctly, the chip will either RESET or erase all key information. If the RESET or erase circuitry has been destroyed, then this unit will not function, thus thwarting an attacker.

The destination of the RESET and Erase line and associated circuitry is very context sensitive. It needs to be protected in much the same way as the individual tamper tests. There is no point generating a RESET pulse if the attacker can simply cut the wire leading to the RESET circuitry. The actual implementation will depend very much on what is to be cleared at RESET, and how those items are cleared.

Finally, Figure 19 shows how the Tamper Lines cover the noise generator circuitry of the chip. The generator and NOT gate are on one level, while the Tamper Detect Lines run on a level above the generator.

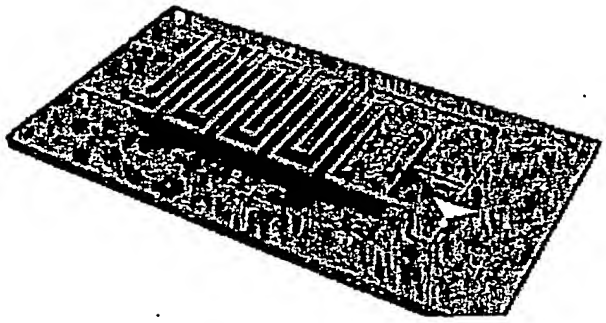


Figure 19. Tamper Detection Lines Cover the Noise Generator

16.1.6 Protected memory with tamper detection

It is not enough to simply store secret information or program code in Flash memory. The Flash memory and RAM must be protected from an attacker who would attempt to modify (or set) a particular bit of program code or key information. The mechanism used must conform to being used in the Tamper Detection Circuitry (described above).

The first part of the solution is to ensure that the Tamper Detection Line passes directly above each Flash or RAM bit. This ensures that an attacker cannot probe the contents of Flash or RAM. A breach of the covering wire is a break in the Tamper Detection Line. The breach causes the Erase signal to be set, thus deleting any contents of the memory. The high frequency noise on the Tamper Detection Line also obscures passive observation.

The second part of the solution for Flash is to use multi-level data storage, but only to use a subset of those multiple levels for valid bit representations. Normally, when multi-level Flash storage is used, a single floating gate holds more than one bit. For example, a 4-voltage-state transistor can represent two bits. Assuming a minimum and maximum voltage representing 00 and 11 respectively, the two middle voltages represent 01 and 10. In the authentication chip, we can use the two middle voltages to represent a single bit, and consider the two extremes to be invalid states. If an attacker attempts to force the state of a bit one way or the other by closing or cutting the gate's circuit, an invalid voltage (and hence invalid state) results.

The second part of the solution for RAM is to use a parity bit. The data part of the register can be checked against the parity bit (which will not match after an attack).

The bits coming from Flash and RAM can therefore be validated by a number of test units (one per bit) connected to the common Tamper Detection Line. The Tamper Detection circuitry would be the first circuitry the data passes through (thus stopping an attacker from cutting the data lines).

While the multi-level Flash protection is enough for non-secret information, such as program code, R, and MinTicks, it is not sufficient for protecting K_1 and K_2. If an attacker adds electrons to a gate (see Section 5.7.2.15 on page 26) representing a single bit of K_1, and the chip boots up yet doesn't activate the Tamper Detection Line, the key bit must have been a 0. If it does activate the Tamper Detection Line, it must have been a 1. For this

reason, all other non-volatile memory can activate the Tamper Detection Line, but K_1 and K_2 must not. Consequently Checksum is used to check for tampering of K_1 and K_2. A signature of the expanded form of K_1 and K_2 (i.e. 320 bits instead of 160 bits for each of K_1 and K_2) is produced, and the result compared against the Checksum. Any non-match causes a clear of all key information.

16.1.7 Boot circuitry for loading program code

Program code should be kept in multi-level Flash instead of ROM, since ROM is subject to being altered in a non-testable way. A boot mechanism is therefore required to load the program code into Flash memory (Flash memory is in an indeterminate state after manufacture).

The boot circuitry must not be in ROM - a small state-machine would suffice. Otherwise the boot code could be modified in an undetectable way.

The boot circuitry must erase all Flash memory, check to ensure the erasure worked, and then load the program code. Flash memory must be erased before loading the program code. Otherwise an attacker could put the chip into the boot state, and then load program code that simply extracted the existing keys. The state machine must also check to ensure that all Flash memory has been cleared (to ensure that an attacker has not cut the Erase line) before loading the new program code.

The loading of program code must be undertaken by the secure Programming Station before secret information (such as keys) can be loaded. This step must be undertaken as the first part of the programming process.

16.1.8 Special implementation of FETs for key data paths

The normal situation for FET implementation for the case of a CMOS Inverter (which involves a pMOS transistor combined with an nMOS transistor) as shown in Figure 20:

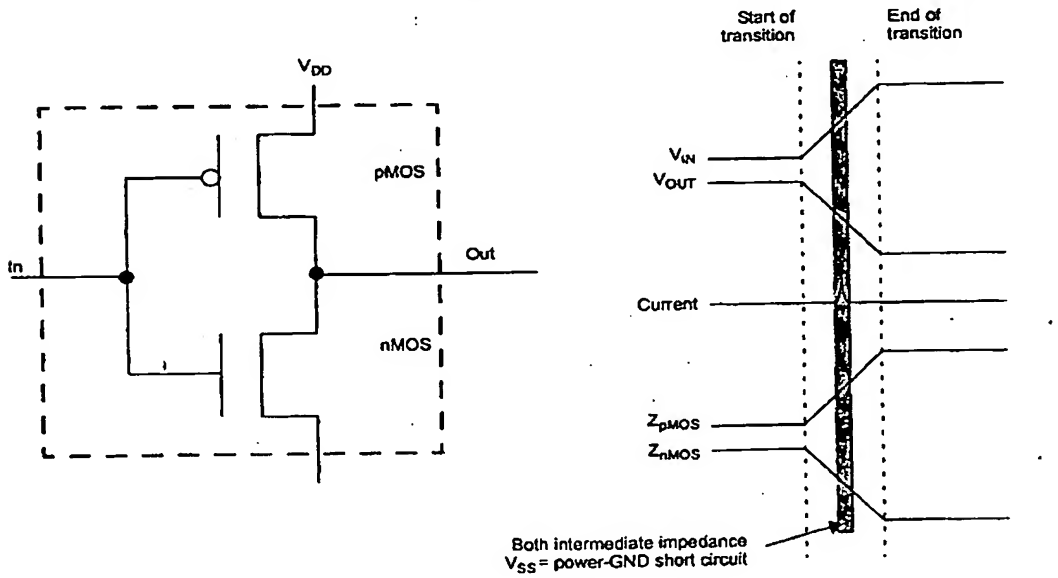


Figure 20. Normal FET Implementation of CMOS Inverter

During the transition, there is a small period of time where both the nMOS transistor and the pMOS transistor have an intermediate resistance. The resultant power-ground short circuit causes a temporary increase in the current, and in fact accounts for the majority of current consumed by a CMOS device. A small amount of infrared light is emitted during the short circuit, and can be viewed through the silicon substrate (silicon is transparent to infrared light). A small amount of light is also emitted during the charging and discharging of the transistor gate capacitance and transmission line capacitance.

For circuitry that manipulates secret key information, such information must be kept hidden. An alternative non-flashing CMOS implementation should therefore be used for all data paths that manipulate the key or a partially calculated value that is based on the key.

The use of two non-overlapping clocks $\phi 1$ and $\phi 2$ can provide a non-flashing mechanism. $\phi 1$ is connected to a second gate of all nMOS transistors, and $\phi 2$ is connected to a second gate of all pMOS transistors. The transition can only take place in combination with the clock. Since $\phi 1$ and $\phi 2$ are non-overlapping, the pMOS and nMOS transistors will not have a simultaneous intermediate resistance. The setup is shown in Figure 21:

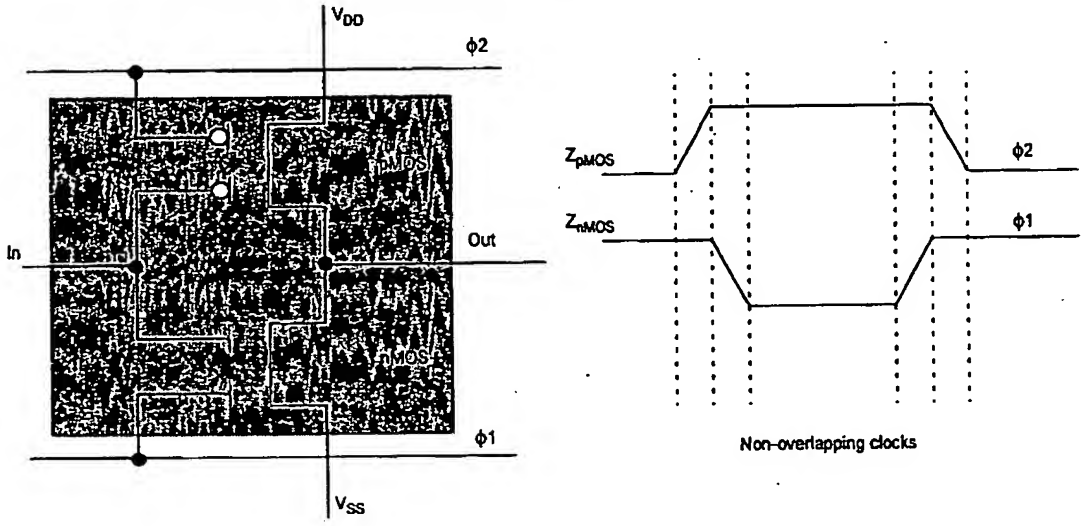


Figure 21. Non-flashing CMOS

Finally, regular CMOS inverters can be positioned near critical non-Flashing CMOS components. These inverters should take their input signal from the Tamper Detection Line above. Since the Tamper Detection Line operates multiple times faster than the regular operating circuitry, the net effect will be a high rate of light-bursts next to each non-Flashing CMOS component. Since a bright light overwhelms observation of a nearby faint light, an observer will not be able to detect what switching operations are occurring in the chip proper. These regular CMOS inverters will also effectively increase the amount of circuit noise, reducing the SNR and obscuring useful EMI.

There are a number of side effects due to the use of non-Flashing CMOS:

- The effective speed of the chip is reduced by twice the rise time of the clock per clock cycle. This is not a problem for an authentication chip.
- The amount of current drawn by the non-Flashing CMOS is reduced (since the short circuits do not occur). However, this is offset by the use of regular CMOS inverters.

- Routing of the clocks increases chip area, especially since multiple versions of $\phi 1$ and $\phi 2$ are required to cater for different levels of propagation. The estimation of chip area is double that of a regular implementation.
- Design of the non-Flashing areas of the authentication chip are slightly more complex than to do the same with a with a regular CMOS design. In particular, standard cell components cannot be used, making these areas full custom. This is not a problem for something as small as an authentication chip, particularly when the entire chip does not have to be protected in this manner.

16.1.9 Connections in polysilicon layers where possible

Wherever possible, the connections along which the key or secret data flows, should be made in the polysilicon layers. Where necessary, they can be in metal 1, but must never be in the top metal layer (containing the Tamper Detection Lines).

16.1.10 OverUnderPower Detection Unit

Each authentication chip requires an OverUnderPower Detection Unit to prevent Power Supply Attacks. An OverUnderPower Detection Unit detects power glitches and tests the power level against a Voltage Reference to ensure it is within a certain tolerance. The Unit contains a single Voltage Reference and two comparators. The OverUnderPower Detection Unit would be connected into the RESET Tamper Detection Line, thus causing a RESET when triggered.

A side effect of the OverUnderPower Detection Unit is that as the voltage drops during a power-down, a RESET is triggered, thus erasing any work registers.

16.1.11 No test circuitry

Test hardware on an authentication chip could very easily introduce vulnerabilities. As a result, the authentication chip should not contain any BIST or scan paths.

The authentication chip *must therefore be testable with external test vectors*. This should be possible since the authentication chip is not complex.

16.1.12 Transparent epoxy packaging

The authentication chip needs to be packaged in transparent epoxy so it can be photo-imaged by the programming station to prevent Trojan horse attacks. The transparent packaging does not compromise the security of the authentication chip since an attacker can fairly easily remove a chip from its packaging. For more information see Section 16.2.20 on page 108 and [86].

16.2 Resistance To Physical Attacks

While this chapter only describes manufacture in general terms (since this document does not cover a specific implementation of a Protocol C1 authentication chip), we can still make some observations about such a chip's resistance to physical attack. A description of the general form of each physical attack can be found in Section 5.7.2 on page 23.

16.2.1 Reading ROM

This attack depends on the key being stored in an addressable ROM. Since each authentication chip stores its authentication keys in internal Flash memory and not in an addressable ROM, this attack is irrelevant.

16.2.2 Reverse engineering the chip

Reverse engineering a chip is only useful when the security of authentication lies in the algorithm alone. However our authentication chips rely on a secret key, and not in the secrecy of the algorithm. Our authentication algorithm is, by contrast, public, and in any case, an attacker of a high volume consumable is assumed to have been able to obtain detailed plans of the internals of the chip.

In light of these factors, reverse engineering the chip itself, as opposed to the stored data, poses no threat.

16.2.3 Usurping the authentication process

There are several forms this attack can take, each with varying degrees of success. In all cases, it is assumed that a clone manufacturer will have access to both the System and the consumable designs.

An attacker may attempt to build a chip that tricks the System into returning a valid code instead of generating an authentication code. This attack is not possible for two reasons. The first reason is that System authentication chips and Consumable authentication chips, although physically identical, are programmed differently. In particular, the RD opcode and the RND opcode are the same, as are the WR and TST opcodes. A System authentication Chip cannot perform a RD command since every call is interpreted as a call to RND instead. The second reason this attack would fail is that separate serial data lines are provided from the System to the System and Consumable authentication chips. Consequently neither chip can see what is being transmitted to or received from the other.

If the attacker builds a clone chip that ignores WR commands (which decrement the consumable remaining), Protocol C1 ensures that the subsequent RD will detect that the WR did not occur. The System will therefore not go ahead with the use of the consumable, thus thwarting the attacker. The same is true if an attacker simulates loss of contact before authentication - since the authentication does not take place, the use of the consumable doesn't occur.

An attacker is therefore limited to modifying each System in order for clone consumables to be accepted (see Section 16.2.4 on page 103 for details of resistance this attack).

16.2.4 Modification of system

The simplest method of modification is to replace the System's authentication chip with one that simply reports success for each call to TST. This can be thwarted by System calling TST several times for each authentication, with the first few times providing false values, and expecting a fail from TST. The final call to TST would be expected to succeed. The number of false calls to TST could be determined by some part of the returned result from RD or from the system clock. Unfortunately an attacker could simply rewire System so that the new System clone authentication chip can monitor the returned result from the consumable chip or clock. The clone System authentication chip would only return success when that monitored value is presented to its TST function. Clone consumables could then return any value as the hash result for RD, as the clone System chip would declare that value valid. There is therefore no point for the System to call the System authentication chip multiple times, since a rewiring attack will only work for the System that has been rewired, and not for all Systems.

A similar form of attack on a System is a replacement of the System ROM. The ROM program code can be altered so that the Authentication never occurs. There is nothing that can

be done about this, since the System remains in the hands of a consumer. Of course this would void any warranty, but the consumer may consider the alteration worthwhile if the clone consumable were extremely cheap and more readily available than the original item.

The System/consumable manufacturer must therefore determine how likely an attack of this nature is. Such a study must include given the pricing structure of Systems and Consumables, frequency of System service, advantage to the consumer of having a physical modification performed, and where consumers would go to get the modification performed.

The likelihood of physical alteration increases with the perceived artificiality of the consumable marketing scheme. It is one thing for a consumable to be protected against clone manufacturers. It is quite another for a consumable's market to be protected by a form of exclusive licensing arrangement that creates what is viewed by consumers as artificial markets. In the former case, owners are not so likely to go to the trouble of modifying their system to allow a clone manufacturer's goods. In the latter case, consumers are far more likely to modify their System. A case in point is DVD. Each DVD is marked with a region code, and will only play in a DVD player from that region. Thus a DVD from the USA will not play in an Australian player, and a DVD from Japan, Europe or Australia will not play in a USA DVD player. Given that certain DVD titles are not available in all regions, or because of quality differences, pricing differences or timing of releases, many consumers have had their DVD players modified to accept DVDs from *any* region. The modification is usually simple (it often involves soldering a single wire), voids the owner's warranty, and often costs the owner some money. But the interesting thing to note is that the change is not made so the consumer can use clone consumables - the consumer will still only buy real consumables, but from different regions. The modification is performed to remove what is viewed as an artificial barrier, placed on the consumer by the movie companies. In the same way, a System/Consumable scheme that is viewed as unfair will result in people making modifications to their Systems.

The limit case of modifying a system is for a clone manufacturer to provide a completely clone System which takes clone consumables. This may be simple competition or violation of patents. Either way, it is beyond the scope of the authentication chip and depends on the technology or service being cloned.

16.2.5 Direct viewing of chip operation by conventional probing

In order to view the chip operation, the chip must be operating. However, the Tamper Prevention and Detection circuitry covers those sections of the chip that process or hold the key. It is not possible to view those sections through the Tamper Prevention lines.

An attacker cannot simply slice the chip past the Tamper Prevention layer, for this will break the Tamper Detection Lines and cause an erasure of all keys at power-up. Simply destroying the erasure circuitry is not sufficient, since the multiple ChipOK bits (now all 0) feeding into multiple units within the authentication chip will cause the chip's regular operating circuitry to stop functioning.

To set up the chip for an attack, then, requires the attacker to delete the Tamper Detection lines, stop the Erasure of Flash memory, and somehow rewire the components that relied on the ChipOK lines. Even if all this could be done, the act of slicing the chip to this level will most likely destroy the charge patterns in the non-volatile memory that holds the keys, making the process fruitless.

16.2.6 Direct viewing of the non-volatile memory

If the authentication chip were sliced so that the floating gates of the Flash memory were exposed, without discharging them, then the keys could probably be viewed directly using an STM or SKM.

However, slicing the chip to this level without discharging the gates is probably impossible. Using wet etching, plasma etching, ion milling, or chemical mechanical polishing will almost certainly discharge the small charges present on the floating gates. This is true of regular Flash memory, but even more so of multi-level Flash memory.

16.2.7 Viewing the light bursts caused by state changes

All sections of circuitry that manipulate secret key information are implemented in the non-Flashing CMOS described above. This prevents the emission of the majority of light bursts. Regular CMOS inverters placed in close proximity to the non-Flashing CMOS will hide any faint emissions caused by capacitor charge and discharge. The inverters are connected to the Tamper Detection circuitry, so they change state many times (at the high clock rate) for each non-Flashing CMOS state change.

16.2.8 Viewing the keys using an SEPM

An SEPM attack can be simply thwarted by adding a metal layer to cover the circuitry. However an attacker could etch a hole in the layer, so this is not an appropriate defense.

The Tamper Detection circuitry described above will shield the signal as well as cause circuit noise. The noise will actually be a greater signal than the one that the attacker is looking for. If the attacker attempts to etch a hole in the noise circuitry covering the protected areas, the chip will not function, and the SEPM will not be able to read any data.

An SEPM attack is therefore fruitless.

16.2.9 Monitoring EMI

The Noise Generator described above will cause circuit noise. The noise will interfere with other electromagnetic emissions from the chip's regular activities and thus obscure any meaningful reading of internal data transfers.

16.2.10 Viewing I_{dd} fluctuations

The solution against this kind of attack is to decrease the SNR in the I_{dd} signal. This is accomplished by increasing the amount of circuit noise and decreasing the amount of signal.

The Noise Generator circuit (which also acts as a defense against EMI attacks) will also cause enough state changes each cycle to obscure any meaningful information in the I_{dd} signal.

In addition, the special Non-Flashing CMOS implementation of the key-carrying data paths of the chip prevents current from flowing when state changes occur. This has the benefit of reducing the amount of signal.

16.2.11 Differential fault analysis

Differential fault bit errors are introduced in a non-targeted fashion by ionization, microwave radiation, and environmental stress. The most likely effect of an attack of this nature is a change in Flash memory (causing an invalid state) or RAM (bad parity). Invalid states and bad parity are detected by the Tamper Detection Circuitry, and cause an erasure of the key.

Since the Tamper Detection Lines cover the key manipulation circuitry, any error introduced in the key manipulation circuitry will be mirrored by an error in a Tamper Detection Line. If the Tamper Detection Line is affected, the chip will either continually RESET or simply erase the key upon a power-up, rendering the attack fruitless.

Rather than relying on a non-targeted attack and hoping that "just the right part of the chip is affected in just the right way", an attacker is better off trying to introduce a targeted fault (such as overwrite attacks, gate destruction etc.). For information on these targeted fault attacks, see the relevant sections below.

16.2.12 Clock glitch attacks

The Clock Filter (described above) eliminates the possibility of clock glitch attacks.

16.2.13 Power supply attacks

The OverUnderPower Detection Unit (described above) eliminates the possibility of power supply attacks.

16.2.14 Overwriting ROM

Authentication chips store program code, keys and secret information in Flash memory, and not in ROM. This attack is therefore not possible.

16.2.15 Modifying EEPROM/Flash

Authentication chips store program code, keys and secret information in multi-level Flash memory. However the Flash memory is covered by two Tamper Prevention and Detection Lines. If either of these lines is broken (in the process of destroying a gate via a laser-cutter) the attack will be detected on power-up, and the chip will either RESET (continually) or erase the keys from Flash memory. This process is described in Section 16.1.6 on page 99.

Even if an attacker is able to somehow access the bits of Flash and destroy or short out the gate holding a particular bit, this will force the bit to have no charge or a full charge. These are both invalid states for the authentication chip's usage of the multi-level Flash memory (only the two middle states are valid). When that data value is transferred from Flash, detection circuitry will cause the Erasure Tamper Detection Line to be triggered - thereby erasing the remainder of Flash memory and RESETing the chip. This is true for program code, and non-secret information. As key data is read from multi-level flash memory, it is not imediately checked for validity (otherwise information about the key is given away). Instead, a specific key validation mechanism is used to protect the secret key information.

An attacker could theoretically etch off the upper levels of the chip, and deposit enough electrons to change the state of the multi-level Flash memory by 1/3. If the beam is high enough energy it might be possible to focus the electron beam through the Tamper Prevention and Detection Lines. As a result, the authentication chip must perform a validation of

the keys before replying to the Random, Test or Random commands. The SHA-1 algorithm must be run on the keys, and the results compared against an internal checksum value. This gives an attacker a 1 in 2^{160} chance of tricking the chip, which is the same chance as guessing either of the keys.

A Modify EEPROM/Flash attack is therefore fruitless.

16.2.16 Gate destruction attacks

Gate Destruction Attacks rely on the ability of an attacker to modify a single gate to cause the chip to reveal information during operation. However any circuitry that manipulates secret information is covered by one of the two Tamper Prevention and Detection lines. If either of these lines is broken (in the process of destroying a gate) the attack will be detected on power-up, and the chip will either RESET (continually) or erase the keys from Flash memory.

To launch this kind of attack, an attacker must first reverse-engineer the chip to determine which gate(s) should be targeted. Once the location of the target gates has been determined, the attacker must break the covering Tamper Detection line, stop the Erasure of Flash memory, and somehow rewire the components that rely on the ChipOK lines. Rewiring the circuitry cannot be done without slicing the chip, and even if it could be done, the act of slicing the chip to this level will most likely destroy the charge patterns in the non-volatile memory that holds the keys, making the process fruitless.

16.2.17 Overwrite attack

An overwrite attack relies on being able to set individual bits of the key without knowing the previous value. It relies on probing the chip, as in the conventional probing attack and destroying gates as in the gate destruction attack. Both of these attacks (as explained in their respective sections), will not succeed due to the use of the Tamper Prevention and Detection Circuitry and ChipOK lines.

However, even if the attacker is able to somehow access the bits of Flash and destroy or short out the gate holding a particular bit, this will force the bit to have no charge or a full charge. These are both invalid states for the authentication chip's usage of the multi-level Flash memory (only the two middle states are valid). When that data value is transferred from Flash detection circuitry will cause the Erasure Tamper Detection Line to be triggered - thereby erasing the remainder of Flash memory and RESETing the chip. In the same way, a parity check on tampered values read from RAM will cause the Erasure Tamper Detection Line to be triggered.

An overwrite attack is therefore fruitless.

16.2.18 Memory remanence attack

Any working registers or RAM within the authentication chip may be holding part of the authentication keys when power is removed. The working registers and RAM would continue to hold the information for some time after the removal of power. If the chip were sliced so that the gates of the registers/RAM were exposed, without discharging them, then the data could probably be viewed directly using an STM.

The first defense can be found above, in the description of defense against power glitch attacks. When power is removed, all registers and RAM are cleared, just as the RESET condition causes a clearing of memory.

The chances then, are less for this attack to succeed than for a reading of the Flash memory. RAM charges (by nature) are more easily lost than Flash memory. The slicing of the chip to reveal the RAM will certainly cause the charges to be lost (if they haven't been lost simply due to the memory not being refreshed and the time taken to perform the slicing).

This attack is therefore fruitless.

16.2.19 Chip theft attack

There are distinct phases in the lifetime of an authentication chip. Chips can be stolen when at any of these stages:

- After manufacture, but before programming of key
- After programming of key, but before programming of state data
- After programming of state data, but before insertion into the consumable or system
- After insertion into the system or consumable

A theft in between the chip manufacturer and programming station would only provide the clone manufacturer with blank chips. This merely compromises the sale of authentication chips, not anything authenticated by the authentication chips. Since the programming station is the only mechanism with consumable and system product keys, a clone manufacturer would not be able to program the chips with the correct key. Clone manufacturers would be able to program the blank chips for their own Systems and Consumables, but it would be difficult to place these items on the market without detection.

The second form of theft can only happen in a situation where an authentication chip passes through two or more distinct programming phases. This is possible, but unlikely. In any case, the worst situation is where no state data has been programmed, so all of M is read/write. If this were the case, an attacker could attempt to launch an adaptive chosen text attack on the chip. The HMAC-SHA1 algorithm is resistant to such attacks. For more information see Section 14.7 on page 67.

The third form of theft would have to take place in between the programming station and the installation factory. The authentication chips would already be programmed for use in a particular system or for use in a particular consumable. The only use these chips have to a thief is to place them into a clone System or clone Consumable. Clone systems are irrelevant - a cloned System would not even require an authentication chip. For clone Consumables, such a theft would limit the number of cloned products to the number of chips stolen. A single theft should not create a supply constant enough to provide clone manufacturers with a cost-effective business.

The final form of theft is where the System or Consumable itself is stolen. When the theft occurs at the manufacturer, physical security protocols must be enhanced. If the theft occurs anywhere else, it is a matter of concern only for the owner of the item and the police or insurance company. The security mechanisms that the authentication chip uses assume that the consumables and systems are in the hands of the public. Consequently, having them stolen makes no difference to the security of the keys.

16.2.20 Trojan horse attack

A Trojan horse attack involves an attacker inserting a fake authentication chip into the programming station and retrieving the same chip after it has been programmed with the secret key information. The difficulty of these two tasks depends on both logical and physical security, but is an expensive attack - the attacker has to manufacture a false authentication chip, and it will only be useful where the effort is worth the gain. For example,

obtaining the secret key for a specific car's authentication chip is most likely not worth an attacker's efforts, while the key for a printer's ink cartridge may be very valuable.

The problem arises if the programming station is unable to tell a Trojan horse authentication chip from a real one - which is the problem of authenticating the authentication chip.

One solution to the authentication problem is for the manufacturer to have a programming station attached to the end of the production line. Chips passing the manufacture QA tests are programmed with the manufacturer's secret key information. The chip can therefore be verified by the C1 authentication protocol, and give information such as the expected batch number, serial number etc. The information can be verified and recorded, and the valid chip can then be reprogrammed with the System or Consumable key and state data. An attacker would have to substitute an authentication chip with a Trojan horse programmed with the manufacturer's secret key information and copied batch number data from the removed authentication chip. This is only possible if the manufacturer's secret key is compromised (the key is changed regularly and not known by a human) or if the physical security at the manufacturing plant is compromised at the end of the manufacturing chain.

Even if the solution described were to be undertaken, the possibility of a Trojan horse attack does not go away - it merely is removed to the manufacturer's physical location. A better solution requires no physical security at the manufacturing location.

The preferred solution then, is to use transparent epoxy on the chip's packaging and to image the chip before programming it. Once the chip has been mounted for programming it is in a known fixed orientation. It can therefore be high resolution photo-imaged and X-rayed from multiple directions, and the images compared against "signature" images. Any chip not matching the image signature is treated as a Trojan horse and rejected.

REFERENCES

17 References

[1] Anderson, R, and Kuhn, M., 1997, *Low Cost Attacks on Tamper Resistant Devices*, Security Protocols, Proceedings 1997, LNCS 1361, B. Christianson, B. Crispo, M. Lomas, M. Roe, Eds., Springer-Verlag, pp.125-136.

[2] Anderson, R., and Needham, R.M., *Programming Satan's Computer*, Computer Science Today, LNCS 1000, pp. 426-441.

[3] Atkins, D., Graff, M., Lenstra, A.K., and Leyland, P.C., 1995, *The Magic Words Are Squeamish Ossifrage*, Advances in Cryptology - ASIACRYPT '94 Proceedings, Springer-Verlag, pp. 263-277.

[4] Bains, S., 1997, *Optical schemes tried out in IC test - IBM and Lucent teams take passive and active paths, respectively, to imaging*. EETimes, December 22, 1997.

[5] Bao, F., Deng, R. H., Yan, Y, Jeng, A., Narasimhalu, A.D., Ngair, T., 1997, *Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults*, Security Protocols, Proceedings 1997, LNCS 1361, B. Christianson, B. Crispo, M. Lomas, M. Roe, Eds., Springer-Verlag, pp. 115-124.

[6] Bellare, M., Canetti, R., and Krawczyk. H., 1996, *Keying Hash Functions For Message Authentication*, Advances in Cryptology, Proceedings Crypto'96, LNCS 1109, N. Koblitz, Ed., Springer-Verlag, 1996, pp.1-15. Full version: http://www.research.ibm.com/security/keyed-md5.html

[7] Bellare, M., Canetti, R., and Krawczyk, H., 1996, *The HMAC Construction*, RSA Laboratories CryptoBytes, Vol. 2, No 1, 1996, pp. 12-15.

[8] Bellare, M., Guérin, R., and Rogaway, P., 1995, *XOR MACs: New Methods For Message Authentication Using Finite Pseudorandom Functions*, Advances in Cryptology, Proceedings Crypto'95, LNCS 963, D Coppersmith, Ed., Springer-Verlag, 1995, pp. 15-28.

[9] Blaze, M., Diffie, W., Rivest, R., Schneier, B., Shimomura, T., Thompson, E., Wiener, M., 1996, *Minimal Key Lengths For Symmetric Ciphers To Provide Adequate Commercial Security. A Report By an Ad Hoc Group of Cryptographers and Computer Scientists*, Published on the internet: http://www.livelinks.com/livelinks/bsa/cryptographers.html

[10] Blum, L., Blum, M., and Shub, M., *A Simple Unpredictable Pseudo-random Number Generator*, SIAM Journal of Computing, vol 15, no 2, May 1986, pp 364-383.

[11] Bosselaers, A., and Preneel, B., editors, 1995, *Integrity Primitives for Secure Information Systems: Final Report of RACE Integrity Primitives Evaluation RIPE-RACE 1040*, LNCS 1007, Springer-Verlag, New York.

[12] Brassard, G., 1988, *Modern Cryptography*, a Tutorial, LNCS 325, Springer-Verlag.

[13] Canetti, R., 1997, *Towards Realizing Random Oracles: Hash Functions That Hide All Partial Information, Advances in Cryptology*, Proceedings Crypto'97, LNCS 1294, B. Kaliski, Ed., Springer-Verlag, pp. 455-469.

[14] Cheng, P., and Glenn, R., 1997, *Test Cases for HMAC-MD5 and HMAC-SHA-1*, Network Working Group RFC 2202, http://reference.ncrs.usda.gov/ietf/rfc/2300/rfc2202.htm

[15] Diffie, W., and Hellman, M.E., 1976, *Multiuser Cryptographic Techniques*, AFIPS national Computer Conference, Proceedings '76, pp. 109-112.

[16] Diffie, W., and Hellman, M.E., 1976, *New Directions in Cryptography*, IEEE Transactions on Information Theory, Volume IT-22, No 6 (Nov 1976), pp. 644-654.

[17] Diffie, W., and Hellman, M.E., 1977, *Exhaustive Cryptanalysis of the NBS Data Encryption Standard*, Computer, Volume 10, No 6, (Jun 1977), pp. 74-84.

[18] Dobbertin, H., 1995, *Alf Swindles Ann*, RSA Laboratories CryptoBytes, Volume 1, No 3, p. 5.

[19] Dobbertin, H, 1996, *Cryptanalysis of MD4*, Fast Software Encryption - Cambridge Workshop, LNCS 1039, Springer-Verlag, 1996, pp 53-69.

[20] Dobbertin, H, 1996, *The Status of MD5 After a Recent Attack*, RSA Laboratories CryptoBytes, Volume 2, No 2, pp. 1, 3-6.

[21] Dreifus, H., and Monk, J.T., 1988, *Smart Cards - A Guide to Building and Managing Smart Card Applications*, John Wiley and Sons.

[22] ElGamal, T., 1985, *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, Advances in Cryptography, Proceedings Crypto'84, LNCS 196, Springer-Verlag, pp. 10-18.

[23] ElGamal, T., 1985, *A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms*, IEEE Transactions on Information Theory, Volume 31, No 4, pp. 469-472

[24] Feige, U., Fiat, A, and Shamir, A., 1988, *Zero Knowledge Proofs of Identity*, J Cryptography, Volume 1, pp. 77-904.

[25] Feigenbaum, J., 1992, *Overview of Interactive Proof Systems and Zero-Knowledge*, Contemporary Cryptology - The Science of Information Integrity, G Simmons, Ed., IEEE Press, New York.

[26] FIPS 46-1, 1977, *Data Encryption Standard*, NIST, US Department of Commerce, Washington D.C., Jan 1977.

[27] FIPS 180, 1993, *Secure Hash Standard*, NIST, US Department of Commerce, Washington D.C., May 1993.

[28] FIPS 180-1, 1995, *Secure Hash Standard*, NIST, US Department of Commerce, Washington D.C., April 1995.

[29] FIPS 186, 1994, *Digital Signature Standard*, NIST, US Department of Commerce, Washington D.C., 1994.

[30] Gardner, M., 1977, *A New Kind of Cipher That Would Take Millions of Years to Break*, Scientific American, Vol. 237, No. 8, pp. 120-124.

[31] Girard, P., Roche, F. M., Pistoulet, B., 1986, *Electron Beam Effects on VLSI MOS: Conditions for Testing and Reconfiguration*, Wafer-Scale Integration, G. Saucier and J. Trihle, Eds., Amsterdam.

[32] Girard, P., Pistoulet, B., Valenza, M., and Lorival, R., 1987, *Electron Beam Switching of Floating Gate MOS Transistors*, IFIP International Workshop onWafer Scale International, Brunel University, Sept. 23-25, 1987.

[33] Goldberg, I., and Wagner, D., 1996, *Randomness and the Netscape Browser*, Dr. Dobb's Journal, January 1996.

[34] Guilou, L. G., Ugon, M., and Quisquater, J., 1992, *The Smart Card, Contemporary Cryptology - The Science of Information Integrity*, G Simmons, Ed., IEEE Press, New York.

[35] Gutman, P., 1996, *Secure Deletion of Data From Magnetic and Solid-State Memory*, Sixth USENIX Security Symposium Proceedings (July 1996), pp. 77-89.

[36] Hendry, M., 1997, *Smart Card Security and Applications*, Artech House, Norwood MA.

[37] Holgate, S. A., 1998, *Sensing is Believing*, New Scientist, 15 August 1998, p 20.

[38] Johansson, T., 1997, *Bucket Hashing with a Small Key Size*, Advances in Cryptology, Proceedings Eurocrypt'97, LNCS 1233, W. Fumy, Ed., Springer-Verlag, pp. 149-162.

[39] Kahn, D., 1967, *The Codebreakers: The Story of Secret Writing*, New York: Macmillan Publishing Co.

[40] Kaliski, B., 1991, *Letter to NIST regarding DSS*, 4 Nov 1991.

[41] Kaliski, B., 1998, *New Threat Discovered and Fixed*, RSA Laboratories Web site http://www.rsa.com/rsalabs/pkcs1

[42] Kaliski, B., and Robshaw, M, 1995, *Message Authentication With MD5*, RSA Laboratories CryptoBytes, Volume 1, No 1, pp. 5-8.

[43] Kaliski, B., and Yin, Y.L., 1995, *On Differential and Linear Cryptanalysis of the RC5 Encryption Algorithm*, Advances in Cryptology, Proceedings Crypto '95, LNCS 963, D. Coppersmith, Ed., Springer-Verlag, pp. 171-184.

[44] Klapper, A., and Goresky, M., 1994, *2-Adic Shift Registers*, Fast Software Encryption: Proceedings Cambridge Security Workshop '93, LNCS 809, R. Anderson, Ed., Springer-Verlag, pp. 174-178.

[45] Klapper, A., 1996, *On the Existence of Secure Feedback Registers*, Advances in Cryptology, Proceedings Eurocrypt'96, LNCS 1070, U. Maurer, Ed., Springer-Verlag, pp. 256-267.

[46] Kleiner, K., 1998, *Cashing in on the not so smart cards*, New Scientist, 20 June 1998, p 12.

[47] Knudsen, L.R., and Lai, X., *Improved Differential Attacks on RC5*, Advances in Cryptology, Proceedings Crypto'96, LNCS 1109, N. Koblitz, Ed., Springer-Verlag, 1996, pp.216-228

[48] Knuth, D.E., 1998, *The Art of Computer Programing - Volume 2/ Seminumerical Algorithms*, 3rd edition, Addison-Wesley.

[49] Krawczyk, H., 1995, *New Hash Functions for Message Authentication*, Advances in Cryptology, Proceedings Eurocrypt'95, LNCS 921, L Guillou, J Quisquater, (editors), Springer-Verlag, pp. 301-310.

[50] Krawczyk, H., 199x, *Network Encryption - History and Patents*, internet publication: http://www.cygnus.com/~gnu/netcrypt.html

[51] Krawczyk, H., Bellare, M, Canetti, R., 1997, *HMAC: Keyed Hashing for message Authentication*, Network Working Group RFC 2104, http://reference.ncrs.usda.gov/ietf/rfc/2200/rfc2104.htm

[52] Lai, X., 1992, *On the Design and Security of Block Ciphers*, ETH Series in Information Processing, J.L. Massey (editor), Volume 1, Konstanz: hartung-Gorre Verlag (Zurich).

[53] Lai, X, and Massey, 1991, J.L, *A Proposal for a New Block Encryption Standard*, Advances in Cryptology, Proceedings Eurocrypt'90, LNCS 473, Springer-Verlag, pp. 389-404.

[54] Massey, J.L., 1969, *Shift Register Sequences and BCH Decoding*, IEEE Transactions on Information Theory, IT-15, pp. 122-127.

[55] Mende, B., Noll, L., and Sisodiya, S., 1997, *How Lavarand Works*, Silicon Graphics Incorporated, published on Internet: http://lavarand.sgi.com (also reported in Scientific American, November 1997 p. 18, and New Scientist, 8 November 1997).

[56] Menezes, A. J., van Oorschot, P. C., Vanstone, S. A., 1997, *Handbook of Applied Cryptography*, CRC Press.

[57] Merkle, R.C., 1978, *Secure Communication Over Insecure Channels*, Communications of the ACM, Volume 21, No 4, pp. 294-299.

[58] Montgomery, P. L., 1985, *Modular Multiplication Without Trial Division*, Mathematics of Computation, Volume 44, Number 170, pp. 519-521.

[59] Moreau, T., *A Practical "Perfect" Pseudo-Random Number Generator*, paper submitted to Computers in Physics on February 27 1996, Internet version: http://www.connotech.com/BBS.HTM

[60] Moreau, T., 1997, *Pseudo-Random Generators, a High-Level Survey-in-Progress*, Published on the internet: http://www.cabano.com/connotech/RNG.HTM

[61] NIST, 1994 , *Digital Signature Standard*, NIST ISL Bulletin, online version at http://csrc.ncsl.nist.gov/nistbul/cs194-11.txt

[62] Oehler, M., Glenn, R., 1997, *HMAC-MD5 IP Authentication with Replay Prevention*, Network Working Group RFC 2085, http://reference.ncrs.usda.gov/ietf/rfc/2100/rfc2085.txt

[63] Oppliger, R., 1996, *Authentication Systems For Secure Networks*, Artech House, Norwood MA.

[64] Preneel, B., van Oorschot, P.C., 1996, *MDx-MAC And Building Fast MACs From Hash Functions*, Advances in Cryptology, Proceedings Crypto'95, LNCS 963, D. Coppersmith, Ed., Springer-Verlag, pp. 1-14.

[65] Preneel, B., van Oorschot, P.C., 1996, *On the Security of Two MAC Algorithms*, Advances in Cryptology, Proceedings Eurocrypt'96, LNCS 1070, U. Maurer, Ed., Springer-Verlag, 1996, pp. 19-32.

[66] Preneel, B., Bosselaers, A., Dobbertin, H., 1997, *The Cryptographic Hash Function RIPEMD-160*, CryptoBytes, Volume 3, No 2, 1997, pp. 9-14.

[67] Rankl, W., and Effing, W., 1997, *Smart Card Handbook*, John Wiley and Sons (first published as Handbuch der Chipkarten, Carl Hanser Verlag, Munich, 1995).

[68] Ritter, T., 1991, *The Efficient Generation of Cryptographic Confusion Sequences*, Cryptologia, Volume 15, No 2, pp. 81-139.

[69] Rivest, R.L, 1993, *Dr. Ron Rivest on the Difficulties of Factoring*, Ciphertext: The RSA Newsletter, Vol 1, No 1, pp. 6, 8.

[70] Rivest, R.L., 1991, *The MD4 Message-Digest Algorithm*, Advances in Cryptology, Proceedings Crypto'90, LNCS 537, S. Vanstone, Ed., Springer-Verlag, pp. 301-311.

[71] Rivest, R.L., 1992, *The RC4 Encryption Algorithm*, RSA Data Security Inc. (This document has not been made public).

[72] Rivest, R.L., 1992, *The MD4 Message-Digest Algorithm*, Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.

[73] Rivest, R.L., 1992, *The MD5 Message-Digest Algorithm*, Request for Comments (RFC) 1321, Internet Activities Board, Internet privacy Task Force.

[74] Rivest, R.L., 1995, *The RC5 Encryption Algorithm*, Fast Software Encryption, LNCS 1008, Springer-Verlag, pp. 86-96.

[75] Rivest, R.L., Shamir, A., and Adleman, L.M., 1978, A *Method For Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, Volume 21, No 2, pp. 120-126.

[76] Schneier, S., 1994, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, Fast Software Encryption (December 1993), LNCS 809, Springer-Verlag, pp. 191-204.

[77] Schneier, S., 1995, *The Blowfish Encryption Algorithm - One Year Later*, Dr Dobb's Journal, September 1995.

[78] Schneier, S., 1996, *Applied Cryptography*, Wiley Press.

[79] Schneier, S., 1998, *The Blowfish Encryption Algorithm*, revision date February 25, 1998, http://www.counterpane.com/blowfish.html

[80] Schneier, S., 1998, *The Crypto Bomb is Ticking*, Byte Magazine, May 1998, pp. 97-102.

[81] Schnorr, C.P., 1990, *Efficient Identification and Signatures for Smart Cards*, Advances in Cryptology, Proceedings Eurocrypt'89, LNCS 435, Springer-Verlag, pp. 239-252.

[82] Shamir, A., and Fiat, A., Method, *Apparatus and Article For Identification and Signature*, U.S. Patent number 4,748,668, 31 May 1988.

[83] Shor, W., 1994, *Algorithms for Quantum Computation: Discrete Logarithms and Factoring*, Proc. 35th Symposium. Foundations of Computer Science (FOCS), IEEE Computer Society, Los Alamitos, Calif., 1994.

[84] Silverbrook Research, 2002, 4-3-1-2 *Authentication Chip Technical Reference.*

[85] Silverbrook Research, 2002, *4-3-1-26 Authentication Protocols.*

[86] Silverbrook Research, 2002, *4-3-1-8 QID Requirements Specification.*

[87] Silverbrook Research, 2002, *4-4-1-3 SoPEC Security Overview.*

[88] Simmons, G. J., 1992, *A Survey of Information Authentication*, Contemporary Cryptology - The Science of Information Integrity, G Simmons, Ed., IEEE Press, New York.

[89] Tewksbury, S. K., 1998, *Architectural Fault Tolerance. Integrated Circuit Manufacturability*, Pineda de Gyvez, J., and Pradhan, D. K., Eds., IEEE Press, New York.

[90] TSMC, 2000, *SFC0008_08B9_HE*, 8K × 8 Embedded Flash Memory Specification, Rev 0.1.

[91] Tsudik, G., 1992, *Message Authentication With One-way Hash Functions*, Proceedings of Infocom '92 (Also in Access Control and Policy Enforcement in Internetworks, Ph.D. Dissertation, Computer Science Department, University of Southern California, April 1991).

[92] Vallett. D., Kash, J., and Tsang, J., *Watching Chips Work*, IBM MicroNews, Vol 4, No 1, 1998.

[93] Vazirani, U.V., and Vazirani, V.V., 1984, *Efficient and Secure Random Number Generation*, 25th Symposium. Foundations of Computer Science (FOCS), IEEE Computer Society, 1984, pp. 458-463.

[94] Wagner, D., Goldberg, I., and Briceno, M., 1998, *GSM Cloning*, ISAAC Research Group, University of California, http://www.isaac.cs.berkeley.edu/isaac/gsm-faq.html

[95] Wiener, M.J., 1997, *Efficient DES Key Search - An Update*, RSA Laboratories CryptoBytes, Volume 3, No 2, pp. 6-8.

[96] Zoreda, J.L., and Otón, J.M., 1994, *Smart Cards*, Artech House, Norwood MA.